Stochastic Automata Network Description of Layer-2 Radio Communication Protocols for Performance Analysis

Martin Ostermann

Chair of Communication Networks – Aachen University of Technology
e-mail: ost@comnets.rwth-aachen.de
WWW: http://www.comnets.rwth-aachen.de/˜ost

Abstract — **Stochastic Automata Networks (SANs) have been in use for several years to perform the numerical analysis of certain modelling problems, to substitute or to complement methods like queuing models, Petri-nets, etc. By the use of an SAN descriptor, large Markov chains can be described by small matrices of the size of the underlying single components of an automata network. However, the memory savings have to be paid for by an additional computational overhead. It is often stated that the SAN approach is not suitable for networks with many synchronisation points.**

In this paper a technique is presented which allows SANs to be utilised to analyse tightly coupled networks, like those that describe the behaviour of point to point communication protocols over stochastic channels. We describe how to derive a well defined SAN description from the procedural notation or state definition of a given protocol.

Solving the static state probabilities of the overall system is the key to obtain any performance parameters. Computational issues, i.e. the effective multiplication of a vector with the SAN descriptor, are examined. It is shown that splitting up the transitions into additional sub-steps may have advantages in simplifying the modelling process as well as speeding up the analysis. The technique is presented with an example for analysing the traffic performance of a simple layer-2 (ISO/OSI) protocol over a fading radio channel.

I. Introduction to Stochastic Automata Networks

The basic model utilised by the SAN approach is a group of interacting automata. Good introductions are given in [**?**], [**?**], and [**?**] . The interaction is achieved either by the means of synchronising events or functional transitions. Synchronising events take place when a transition in one automaton is forcing at least one transition in a different automaton. Pure functional transitions are local to one automaton, but their transition probabilities depend on the global state space of all involved automata. Synchronising events may also be functional.

The SAN descriptor is used to describe the corresponding Markov chain of the network. By the use of tensor products (also known as Kronecker products [**?**]), compact descriptions of huge Markov chains can be achieved. The transition matrix of the system can be described as a sum of N_p tensor products

$$P = \sum_{j=1}^{N_P} \otimes_{i=1}^{N} P_j^{(i)},$$

where N denotes the number of automata connected. Note that when considering functional transitions, the matrices $P_j^{(i)}$ describe functions. While P is a stochastic matrix, the individual matrices $P_j^{(i)}$ do not need to be stochastic matrices themselves. N_P increases linearly with the number of synchronising events [**?**]. This may appear to be prohibitive in case of tightly coupled networks, but we will see that most coupling can be described by the use of functions instead. Also, the matrices of those additional terms usually are sparse, a fact that can be exploited for efficiency.

Quite rigorous restrictions on the transitions described by an SAN are necessary to avoid ambiguities resulting in nondeterminism [**?**]. This paper proposes a technique to derive a well defined SAN descriptor from a procedural notation or state machine definition of the protocol to be examined.

A. Local Transitions

Let us first examine the transition matrices of two independent automata with two states each:

$$P^{(1)} = \begin{pmatrix} 1-\lambda_1 & \lambda_1 \\ \lambda_2 & 1-\lambda_2 \end{pmatrix},$$

$$P^{(2)} = \begin{pmatrix} 1-\mu_1 & \mu_1 \\ \mu_2 & 1-\mu_2 \end{pmatrix}.$$

The transition matrix of the Markov chain that describes the combined global system is easily derived and is shown as P_g in Table 1. This result can also be expressed by using tensor algebra as

$$P_g = P^{(1)} \otimes P^{(2)}.$$

B. Synchronising Events

If, for example, the second automaton is forced into state 1 whenever the first automaton switches from state 2 to state 1 (with the probability λ_2), this is called a synchronising event and the automata are no longer independent. We get the matrix P_g', which is also shown in Table 1.

This can be expressed as $P_g' = P_l^{(1)\prime} \otimes P_l^{(2)\prime} + P_{e_1}$ with

$$P_l^{(1)\prime} = \begin{pmatrix} 1-\lambda_1 & \lambda_1 \\ 0 & 1-\lambda_2 \end{pmatrix},$$

$$P_l^{(2)\prime} = \begin{pmatrix} 1-\mu_1 & \mu_1 \\ \mu_2 & 1-\mu_2 \end{pmatrix},$$

and the synchronising event given by

Table 1: Example matrices without (P_g) and with (P'_g) a synchronising event.

$$P_{g=}\begin{pmatrix} (1-\lambda_1)(1-\mu_1) & (1-\lambda_1)\mu_1 & \lambda_1(1-\mu_1) & \lambda_1\mu_1 \\ (1-\lambda_1)\mu_2 & (1-\lambda_1)(1-\mu_2) & \lambda_1\mu_2 & \lambda_1(1-\mu_2) \\ \lambda_2(1-\mu_1) & \lambda_2\mu_1 & (1-\lambda_2)(1-\mu_1) & (1-\lambda_2)\mu_1 \\ \lambda_2\mu_2 & \lambda_2(1-\mu_2) & (1-\lambda_2)\mu_2 & (1-\lambda_2)(1-\mu_2) \end{pmatrix}$$

$$P'_g = \begin{pmatrix} (1-\lambda_1)(1-\mu_1) & (1-\lambda_1)\mu_1 & \lambda_1(1-\mu_1) & \lambda_1\mu_1 \\ (1-\lambda_1)\mu_2 & (1-\lambda_1)(1-\mu_2) & \lambda_1\mu_2 & \lambda_1(1-\mu_2) \\ \lambda_2 & 0 & (1-\lambda_2)(1-\mu_1) & (1-\lambda_2)\mu_1 \\ \lambda_2 & 0 & (1-\lambda_2)\mu_2 & (1-\lambda_2)(1-\mu_2) \end{pmatrix}$$

$$P_{e_1} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \lambda_2 & 0 & 0 & 0 \\ \lambda_2 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ \lambda_2 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

The description has been split into a so-called local part $\otimes_{i=1}^{N} P_l^{(i)}$ and the synchronising event P_{e_1}. When more than one synchronising event is present, each is described by its own tensor product, whereas the local part can always be described by a single tensor product. The local part describes all transitions which are not induced or prohibited by other automata. It may still contain functional dependencies.

Each synchronising event is described by a tensor product of matrices, containing one matrix for each automaton in the system. One automaton is said to "own" the synchronising event[1]. It performs a local transition with a probability described in its matrix. The other matrices can be considered as routing matrices. They often have a distinctive form. In case of the forced setting of a specific state, this is a sparse matrix with a single column of unity entries. The matrices of the automata which are not involved in the synchronising event are the identity matrices I_{n_i}. The SAN needs to be well defined so that no synchronising events interfere with each other [**?**].

We suggest to achieve that by restraining the owning of synchronising events that control a certain automaton to a single automaton (See Section II.D). This ensures that there will be no active transitions that contradict each other, and there will be no aggregated behaviour either. As a side effect, the local transitions are eliminated, but they are emulated by automata matrices which appear in a number of synchronising events. While it might appear that this is a rather severe restriction, we will see that the model is still expressive enough, especially since we have not put any restrictions on the use of functions. Also note that an automaton may still control several other automata, even with a single synchronising event.

II. Modelling of Layer-2 Communication Protocols

A. Suitable protocol classes

There are certain constraints on the type of protocols suitable for a discrete time Markov chain analysis.

[1] In the above example, this is the first automaton.

- For example, the number of communication partners must be low, because the state space increases exponentially with the number of automata involved. We need to store the stationary probability vector, so the number of (reachable) states may not exceed our main storage capabilities.
- Also, in order to combine several discrete time Markov chains, they must proceed synchronously. Basically, this implies that the whole system is virtually timed by a common clock.
- In order to reduce the global state space, it is advisable to model as broad as possible without losing accuracy in the performance analysis. In case of fixed packet sizes or fixed slot times in layer-1, the preferred time step would be the duration of one packet or one slot. Protocols which use fixed length PDUs (Protocol Data Units) like ATM (Asynchronous Transfer Mode) and GPRS (General Packet Radio Service) [**?**] thus appear well suited for this approach, too.

Consider a typical synchronous communication protocol. We further assume a slotted behaviour and packets of constant size, such that one packet is transmitted during each time slot. The transmission of the messages does take some time, so that the receiving side cannot react on the information transmitted before the following time-slot. We allow for a full-duplex behaviour, i.e. both sides may transmit and receive simultaneously in one time-slot, as well as for an interleaved scheme, where only one side is transmitting in one time-slot. The transmissions can be disturbed, so that no packet is transmitted to the receiving side. The packet error ratio is described by an independent stochastic Markov process. This model allows for a simple Gilbert-Elliott model as well as for much more complex stochastic error models, like the generative radio channel models described in [**?**]. The behaviour of each side is described by an extended finite state machine (EFSM), i.e. there is a certain control flow that acts on some variables. We also consider the use of timers to allow for time-out alarms.

B. Implementation of protocol components

Following these provisions, we suggest the following implementation rules (see also Fig. 1 for a graphical representation of an example):

Each variable is modelled by a separate automaton, which can be set by a synchronising event of its associated core.

Instead of setting a specific state, the new state may depend on other variables. For example the core might request to copy the state of another variable. It also might ask to add or subtract the value of a different variable, or just to increment or decrement the state of the variable itself. The local transition matrix of a variable is usually the identity matrix.

Timers can be treated as special variables, which in addition perform independent state changes. Their transition matrices primarily contain non-zero entries in a secondary diagonal.

The core of the EFSM is modelled by a single automaton, which may impose synchronising events on the associated variable automata that model the variables of the machine. The actions of the core automaton may depend on the global state space, thus also on the state of the variables. The matrices describing synchronising events of the core often can be obtained manually. In case of more complex protocols, we suggest to use common compilation techniques [**?**].

Layer-1 is modelled by one or more automata that hold the result of a transmission. It acts much like an ordinary variable, but its state is set by a synchronising event controlled from the other side. This synchronising event thus models the transmission, and therefore is functionally dependent on the state of the channel.

The channel is modelled by an independent automaton. Its state determines via functions the transmission probabilities of information exchanged between both sides in layer-1, modelling the packet error ratio.

A time-slot of the model may be subdivided into two or possibly more sub-steps. This allows to separate the descriptions of receiving and sending packets or to separate between the communication to the upper and lower layers. In Section III, we will apply this technique to analyse the traffic performance of a simple layer-2 protocol over a fading radio channel.

C. Derivation of the state transition graph

In the previous section, we have discussed how to obtain the components of the protocol model. We still need to describe the behaviour of each component, especially the actions of the core. In a first step, we need to derive the state transition graph, i.e. a description of the next state of each automaton based on the current state of the other automata.

One way to obtain such a graph is to start with a flow diagram of the protocol code. We then need to modify the decisions on which the selection of branches depends so that they are based on the original states of the variables, instead on intermediate or new states. This can be done by flattening the graph. A graph is flattened in two steps: First we remove joints of branches by duplication of the remaining parts. In a second step, we move the start of branches up to the root of the graph. Each assignment on the way up needs to be duplicated in both branches. Note that we do not need to flatten out the graph by default, it is just a means to simplify the process of obtaining the transition matrices. Sometimes, it is better to leave some branches within a synchronising event, especially if the branch is already based on the original state of a variable and if no other decision depends on its actions. In such cases, its actions (such as the assignment or increment/decrement of variables) are implemented as functional matrices that depend of the result of the branch decision.

D. Derivation of the transition matrices

Within the SAN approach, the actions of each automaton are described by transition matrices. There is one functional matrix for each synchronising event the automaton is involved in.

We use different synchronising events to select between the top level branches of the state graph. We associate this selection with the core automata, which often just consists of a single state. Hence, this can be described by a boolean function which is a translation of the decisions that select the different branches of the flattened state transition graph. A synchronising event is enabled if the boolean function evaluates to one, otherwise it is disabled. Enabling all synchronising events with a single automaton (which is said to "own" the synchronising event, cf. Section I.B) has the advantage that it is easy to verify that the SAN is well defined. This is reduced to checking that one but only one of the boolean functions of the core evaluates to true for each global state.

Derivation of most other transition matrices is easy, too. For example, assume a variable S should be incremented if and only if it is equal to another variable D. Let the dimension of both variables be three. This "`increment_if_equal()`" function has a transition matrix S for S dependent on the state S_D of D is given as[2]:

$$S = \begin{cases} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & if \quad S_D = 0 \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} & if \quad S_D = 1 \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} & if \quad S_D = 2 \end{cases}$$

III. EXAMPLE PROTOCOL

In this section, we give a step by step introduction to the modelling process of a simple layer-2 protocol. We have chosen a sliding window protocol (Automatic Repeat Request, ARQ), but with simplifications to keep the model descriptions short for this educational purpose. We omitted timers, and we reduced the number of different types of PDUs to two, positive or negative acknowledge with a data field. If there is no new data to be transmitted, the repetition of the last valid data and sequence number indicates this. However, we also omitted a traffic generator, thus assuming continuous traffic.

The following Sections III.A and III.B describe one instance of the two communication partners A and B, that together with a channel model describe the complete scenario. The channel is modelled by a discrete time Gilbert-Elliott

[2]We assume modulo arithmetic.

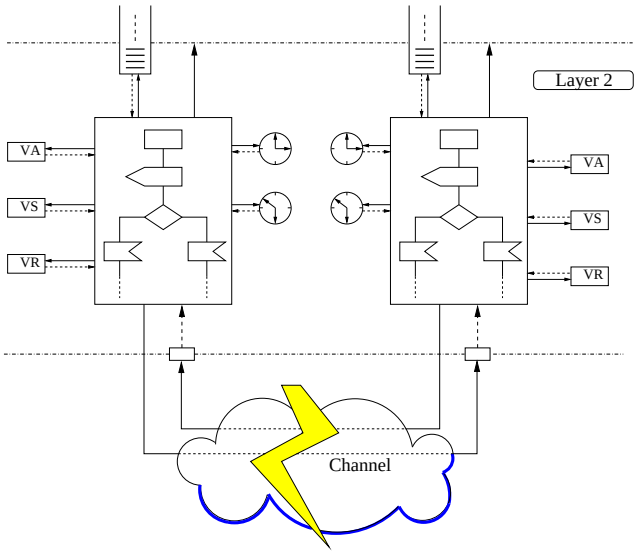

Fig. 1: ARQ protocol example scenario. We see queues, timers, and variables controlled by two core automata implementing the EFSM specification of the communication protocol instances. Dashed arrows express functional dependencies, while solid ones refer to synchronising events.

model. This automaton, which has thus two states, also models the transport in layer-1 dependent on the state of the channel. Its effect is modelled by an additional transition, which acts in between the time-slots of the two communication instances and either transports the PDUs or indicates an error. The actions of the two instances are interleaved, which means that one instance gets an acknowledge for a sent packet before the start of the next cycle. Hence, each cycle consists of two time-slots with two sub-steps each:

1. Instance A acts.
2. A packet is transmitted (or disturbed) from A to B.
3. Instance B acts.
4. A packet is transmitted (or disturbed) from B to A.

A. Verbal description

Each PDU is identified by the packet sequence number P, which identifies the data content. An acknowledge field Q contains the sequence number of the last packet correctly received. The type field of the PDU indicates whether this is a positive or a negative acknowledge. A positive acknowledge states that the instance issuing the acknowledge has received this as the last valid packet. It is to notify the sending party that its packet has been correctly transmitted. A negative acknowledge states that an out of order packet has been received, and it requests to resend all packets that followed the sequence number in the acknowledge field.

Each protocol instance keeps three variables VS, VA, and VR. VS holds the sequence number of the last packet sent. VA holds the value of the last received acknowledge. Thus VS and VA keep the status about transmitted data. VR keeps the status of the received data. It contains the sequence number of the next expected packet.

The difference between VS and VA thus describes the amount of packets that have been transmitted, but not acknowledged yet. The *send window size* (WS) is a system parameter that determines whether a new packet can be send, i.e. if the sequence number can be incremented. This is the case if the difference between VS and VA is less than the send window size.

B. Procedural description

We continue by giving a more formal definition in a programming language similar to C. In each time-slot, the receive part is executed before the transmit part. We omitted the part which is responsible to handle the data content of the packet, since it is not needed for a performance analysis. Instead, we focus on the handling of sequence numbers.

```
int VS, VA, VR;
int Q, P;
enum { DATA, REJ, ERR } T_in, T_out;
```

The variables Q, P, and T_in take the values of the received packet. T_out is a temporary variable to describe the type of the packet to be sent. All actions on variables that hold sequence numbers are considered modulo arithmetic without further notice.

```
receive(&T_in, &P, &Q);
if ( T_in == ERR ) goto TRANSMIT;
if ( P == VR )
   { VR++, T_out = DATA; }
else if ( P < VR )
   { T_out = DATA; }
else /* P > VR */
   { T_out = REJ; }
```

The receive part is responsible to handle the incoming data stream. The acknowledge field of the incoming packet belongs to the outgoing data stream and is thus handled by the transmit part.

First, the variables Q, P, and T_in are initialised with the values of the received packet. In case of an transmission error, the T_in variable contains ERR, and the other variables are invalid. Therefore, in line 2, the receive part is left in case no valid packet was received. Otherwise, the remaining commands set VR and the T_out type dependent on the relation between the received packet sequence number and the expectation stored in VR.

```
TRANSMIT:
if ( T_in == DATA )
   { VA = Q; }
else if ( T_in == REJ )
   { VA = Q, VS = Q}
if ( VS-VA < WS)
   { VS++; }
send( T_out, VS, VR-1)
```

The transmit part first handles acknowledges. If a valid packet was received, the VA variable is set to its new state. If the acknowledge was negative, the VS variable is set back to indicate that these packets need to be resent. After that, it is decided whether it is possible to send a new packet. It is important to recognise that the variable VA might have already be changed prior to this, and the decision is based on its new state.

In any case, a packet is sent at the end. If the VS variable was not incremented, the last packet is retransmitted.

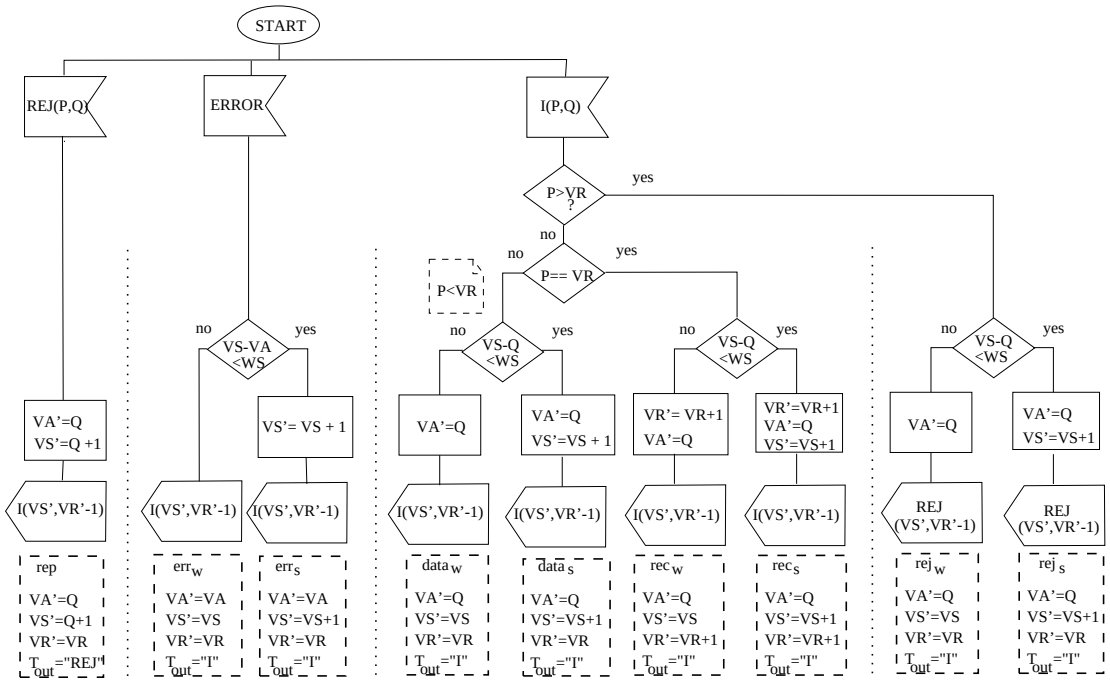

Fig. 2: Flattened state transition graph of the example protocol ("`I`" is called "`DATA`" in the text)

C. Construction of the SAN

We derived the state transition graph as described in Section II.C. The result for one protocol instance is shown in Fig. 2. The upper part of the diagram shows the state transition graph, the lower part denotes the derived transition matrices. In the diagram, a reference to variable `V` denotes the old state, while a reference to `V'` relates to the new state. We needed to implement this twice for each side of the protocol. In addition, we needed an implementation of the channel and of layer-1. In order to save on automata, we overlapped the descriptions: The `T_out` variable of one instance is read by the other instance as the incoming packet type `T_in`, as `VS` becomes `P`, and `VR-1` becomes `Q`. Thus the implementation of layer-1 only needs to modify the `T_out` variable according to its current packet error ratio, which is determined by the state of the channel. According to Fig. 2 we would need nine synchronising events. However, we were able to reduce them down to four by rejoining some of the branches which only differ by the calculations of `VS'` and `VR'`. This is done by introducing two special matrix functions, namely "`VS'=increment_if_possible(VA)`" and "`VR'=increment_if_equal(P)`", but we will not elaborate further on this.

D. Solving the system to perform an analysis

Calculation of the steady state probabilities of the system is the key to obtain a wide range of performance parameters. Simple measures such as throughput or packet loss can be calculated directly by adding up the states where this is possible, multiplied with transition probability for the given event [**?**]. More complex measures, such as the delay distribution, could be obtained by observing the system from a given set of states over the time. This is performed by repetitively multiplying the current state distribution with the transition matrix P.

We implemented and solved the example system for a variable size of four, and a send window size `WS=2`. To obtain the steady state probabilities of this aperiodic system, we needed to solve the system of linear equations described by $(P^T - E)\pi = 0$, $\sum \pi = 1$ [**?**]. The dimension of π is $\dim \pi = \prod_{i=1}^{N} n_i = 2 \cdot (4 \cdot 4 \cdot 4 \cdot 3)^2 = 73728$. We tried to used both the BiCG (BiConjugate Gradient) and QMR (Quasi-Minimal-Residual) algorithm [**?**] to solve the system, but neither did appear to converge well. So we approximated the result vector $\overline{\pi}$ with $\overline{\pi}_i = 1/\dim \pi$ and by multiplying itself a few times with P. Using this approximation of π as a start vector, both the QMR and BiCG algorithms calculate very good approximations for π within few iterations. Repetition of both steps resulted in a result vector accurate to a relative error of $e \approx 10^{-13}$. This took about 15 minutes processor time on a SUN workstation with an UltraSPARC processor at 250 MHz. The simple multiplication $P\pi$ takes about 5 seconds, whilst an iteration of the QMR algorithm takes about 40 seconds. The program needs 14 Megabyte main memory.

Examination of the steady state probabilities revealed that there are only 512 non-zero states left out of the total of 73728. This is due to the fact, that the protocol is designed to keep each instance informed about state of the other one. This also means that a conventional sparse matrix algorithm would perform much better, once a reachability analysis has been performed. This is caused by the fact that the SAN approach is still lacking means to efficiently handle knowledge about non-reachable and transient states. Thus, as long as such means are not implemented, it should be considered to restrict the usage of SANs to be a modelling aid, but to perform the numerical analysis by conventional means.

It is also interesting to note that the 512 non-zero probabilities consist of only 32 unique values. This indicates that exploitation of symmetries would further reduce the size of the linear system that needs to be solved. The SAN ap-

proach should thus be extended to provide means to identify and handle such symmetries.

IV. Computational Issues

A. Avoidance of cyclic dependencies

When solving the steady state probabilities of a huge Markov chain described by an SAN descriptor, it is best to use iterative methods as they do not require the manipulation of the system itself. Instead, the basic operation needed is the multiplication of a vector with the matrix P (or its transpose P^T) described by the SAN [**?**]. A computationally efficient method for this is described in [**?**]. It has been shown that a prerequisite for the avoidance of unnecessary multiplications is the absence of cyclic dependencies of the matrix functions. In this case, the system can be expressed as a sum of generalised matrix tensor products (GMTPs)[**?**].

Splitting up a time-slot into separate sub-steps does not only keep the model simple, but it also helps to resolve such cyclic dependencies. If for each sub-step $k \in [1..n_s]$ we define an extra matrix P_j, we obtain $(((xP_1)P_2)\cdots P_{n_s}) = x\prod_{k=1}^{n_s} P_i = xP$. As each sub-matrix is defined as $P_k = \sum_{i=1}^{N_P} \otimes_{j=1}^{N} P_k^{(j)}$ and tensor algebra is distributive and pseudo commutative [**?**] among matrix multiplication and addition, each tensor product may be reordered in the most efficient way. It is also shown in [**?**] that remaining cyclic dependencies can be removed by further splitting the involved transition matrices P_k into n separate matrices $P_{k,l}$ such that $P_k = \sum_{l=1}^{n} \otimes_{j=1}^{N} P_{k,l}^{(j)}$, thus enabling additional reorder opportunities.

B. Complexity issues

In the literature [**?**, **?**] the evaluation of complexity is usually based on the number of multiplications to be performed. This is probably based on the fact, that the multiplication used to be an expensive operation during the calculation. However, most microprocessors of modern workstations pipeline the multiplication and are able to perform at least one multiplication per cycle. Instead, memory bandwidth becomes a limiting factor, especially if the working set exceeds the capacity of the system cache. In addition, when modelling protocols (or algorithms in general) most transition matrices are deterministic (one 1.0 entry in each row). We optimised on this, thus in our implementation most vector-GMTP products are calculated without the need for multiplications. Thus, it is obvious that the amount of multiplications cannot be the right measure to estimate the runtime complexity, but the same still holds true if the transition matrices are sparse matrices with only a small number of entries per row.

Each in-place evaluation of a normal factor [**?**] performs n_S/n_i ($n_S = \prod_{j=1}^{N} n_j$) shuffle operations where n_i elements are read, modified, and written back to the same n_i locations. The elements are not located consecutively in memory, but lie $\prod_{j=1}^{i} n_j$ locations apart. It depends on n_i and the architecture of the system cache if this operation is as fast as n_S consecutive read and write operations, but usually it will be slower. We nonetheless assume, that the calculation of one normal factor will have the time complexity $O(n_S)$. To multiply a vector with an SAN descriptor, we need to perform such an operation for each functional non-identity automata matrix of all synchronising events. In our experience, this is only one matrix for each automaton in one time step per synchronising event[3]. Thus, when E denotes the number of synchronising events, the total complexity of one matrix vector multiplication with the use of the SAN descriptor amounts to

$$O(\cdot) = N \times E \times \prod_{i=1}^{N} n_i.$$

V. Conclusions

We have introduced a technique for the detailed modelling of layer-2 protocols in a semi-automatic way. The use of SAN descriptors allows us to calculate the steady state probabilities of such models, which is the key to obtain a wide range of performance parameters. Where applicable, it is likely that less abstraction is needed compared to other analysis techniques such as Petri-nets, queuing models, etc. The problem of state space explosion remains the biggest obstacle against broad application. It should be investigated if the SAN approach can be extended to efficiently handle knowledge about non-reachable and transient states.

[3] Plus one or two to seperately model the influence of the channel on the transmission, but this doesn't affect the complexity.