The PEPS Software Tool

Anne Benoit[1], Leonardo Brenner[2], Paulo Fernandes[2],
Brigitte Plateau[1], and William J. Stewart[3]

[1] Laboratoire ID, CNRS-INRIA-INPG-UJF, 51, av. Jean Kuntzmann
38330 Montbonnot Saint-Martin France
{Anne.Benoit, Brigitte.Plateau}@imag.fr
[2] PUCRS, Faculdade de Informática, Av. Ipiranga, 6681
90619-900 Porto Alegre, Brazil
{lbrenner, paulof}@inf.pucrs.br
[3] NCSU, Computer Science Department, Raleigh, NC 27695-8206, USA
Research supported in part by NSF grant ITR-105682
billy@cs.ncsu.edu

Abstract. PEPS is a software package for solving very large Markov models expressed as Stochastic Automata Networks (SAN). The SAN formalism defines a compact storage scheme for the transition matrix of the Markov chain and it uses tensor algebra to handle the basic vector matrix multiplications. Among the diverse application areas to which PEPS may be applied, we cite the areas of computer and communication performance modeling, distributed and parallel systems and finite capacity queueing networks. This paper presents the numerical techniques included in version 2003 of the PEPS software, the basics of its interface and three practical examples.

1 Introduction

Parallel and distributed systems can be modeled as sets of interacting components. Their behavior is usually hard to understand and formal techniques are necessary to check their correctness and predict their performance. A Stochastic Automata Network (SAN) [6, 12] is a formalism to facilitate the modular description of such systems and it allows the automatic derivation of the underlying Markov chain which represents its temporal behavior. Solving this Markov chain for transient or steady state probabilities allows us to derive performance indices. The main difficulties in this process is the complexity of the model and the size of the generated Markov chain.

Several other high-level formalisms have been proposed to help model very large and complex continuous-time Markov chains in a compact and structured manner. For example, queueing networks [9], generalized stochastic Petri nets [1], stochastic reward nets [11] and stochastic activity nets [17] are, thanks to their extensive modeling capabilities, widely used in diverse application domains, and notably in the areas of parallel and distributed systems.

The pioneering work on the use of Kronecker algebra for solving large Markov chains has been conducted in a SAN context. The modular structure of a SAN

model has an impact on the mathematical structure of the Markov chain in that it induces a product form represented by a tensor product. Other formalisms have used this Kronecker technique, as, *e.g.*, stochastic Petri nets [5] and process algebras [10].

The basic idea is to represent the matrix of the Markov chain by means of a tensor (Kronecker) formula, called *descriptor* [2]. This formulation allows very compact storage of the matrix. Moreover, computations can be conducted using only this formulation, thereby saving considerable amounts of memory (as compared to an extensive generation of the matrix). Recently, other formats which considerably reduce the storage cost, such as matrix diagrams [4], have been proposed. They basically follow the same idea: components of the model have independent behaviors and are synchronized at some instants; when they behave independently their properties are stored only once, whatever the state of the rest of the system. Using tensor products, a single small matrix is all that is necessary to describe a large number of transitions. Using matrix diagrams (a representation of the transition matrix as a graph), transitions with the same rate are represented by a single arc. At this time, SAN algorithms use only Kronecker technology, but a SAN model could also be solved using matrix diagrams.

A particular SAN feature is the use of functional rates and probabilities [7]. These are basically state dependent rates, but even if a rate is local for a component (or a subset of components) of the SAN, the functional rate can depend on the entire state of the SAN. It is important to notice that this concept is more general than the usual state dependent concept in queueing networks. In queueing networks, the state dependent service rate is a rate which depends only on the state of the queue itself.

The basic technique to solve SAN models is the so-called *shuffle algorithm* [13], which makes use of the Kronecker product structure, and has acceptable complexity [14]. Since the initial introduction of this algorithm, many improvements have been developed including:

- a reduction in the cost of function evaluation by a different implementation of the shuffle algorithm,
- the reduction of the model product state space and the complexity of the descriptor formula by the automatic grouping of automata,
- the acceleration of the convergence of iterative methods by using preconditioning techniques and projection methods.

These improvements were implemented in the previous version of the software dedicated to solving SAN models, called PEPS - Performance Evaluation of Parallel Systems - (the version PEPS 2000) [2, 6, 7]. The objective of this paper is to present the performance of the new version, PEPS 2003. It is based on the previous version, and offers new features and better performance. Furthermore, the interface has been modified to allow the notion of replication:

- replicas of states when they have the same behavior (typically the inner states of a queue), and
- replicas of automata (typically, identical components of a system).

Another improvement of the PEPS 2003 version results from the function evaluation itself: in previous versions, functions were interpreted while in the new version they are compiled. This paper gives an experimental assessment of this modification. Finally, the shuffle algorithm has been modified in order to work with vectors of the size of the reachable state space [3]. Previously, the shuffle algorithm's main drawback was the use of vectors of the size of the product state space. When the reachable state space is small compared to the product state space (basically less than 50%), the new version of the shuffle algorithm uses less memory and is more efficient.

This paper is divided in sections as follows. Section 2 presents the SAN formalism and PEPS syntax by means of an example. Section 3 briefly presents the algorithms implemented in PEPS. Section 4 presents examples which are used to compute some numerical results showing the performance of PEPS 2003. This improved performance is discussed in the conclusions.

2 A Formalism for Stochastic Automata Networks

In a SAN [7, 13] a system is described as a collection of interacting subsystems. Each subsystem is modeled by a stochastic automaton, and the interaction among automata is described by firing rules for the transitions inside each automaton. The SAN models can be defined on a continuous-time or discrete-time scale. In this paper, attention is focused only on continuous-time models and therefore the occurrence of transitions is described as a rate of occurrence. The concepts presented in this paper can be generalized to discrete-time models, since the theoretical basis of such SAN models has already been established [2]. In this section an informal description of a SAN is presented, and then the textual format to input SAN models in the PEPS 2003 tool is briefly described.

2.1 Informal Description

Each automaton is composed of states, called *local states*, and transitions among them. Transitions on each automaton are labeled with a list of the events that may trigger them. Each event is denoted by its name and its rate (only the name is indicated in the graphical representation of the model). When the occurrence of the same event can lead to different arrival states, a probability of occurrence is assigned to each possible transition. The label on the transition is given as $evt(prob)$, where evt is the event name, and $prob$ is the probability of occurrence. When not explicitly specified, this probability is set to 1.

There are basically two ways in which stochastic automata interact. First, the rate at which an event may occur can be a *function* of the state of other automata. Such rates are called *functional* rates. Rates that are not functional are said to be *constant* rates. The probabilities of occurrence of events can also be functional or constant. Second, an event may involve more than one automaton: the occurrence of such an event triggers transitions in two or more automata at the same time. Such events are called *synchronizing* events. They may have

constant or functional rates. An event which involves only one automaton is said to be a *local* event.

Consider a SAN model with N automata and E events. It is a N-component Markov chain whose components are not necessarily independent (due to the possible presence of functional rates and synchronizing events). A local state of the i-th automaton ($\mathcal{A}^{(i)}$, where $i = 1 \dots N$) is denoted $x^{(i)}$ while the complete set of states for this automaton is denoted $S^{(i)}$, and the cardinality of $S^{(i)}$ is denoted by n_i. A global state for the SAN model is a vector $x = (x^{(1)}, \dots, x^{(N)})$. $\hat{S} = S^{(1)} \times \dots \times S^{(N)}$ is called the product state space, and its cardinality is equal to $\prod_{i=1}^{N} n_i$. The reachable state space of the SAN model is denoted by S; it is generally smaller than the product state space since synchronizing events and functional rates may prevent some states in $\hat{S}$ from being reachable. The set of automata involved with a (local or synchronizing event) e is denoted by O_e. The event e can occur if, and only if, all the automata in O_e are in a local state from which a transition labeled by e can be triggered. When it occurs, all the corresponding transitions are triggered. Notice that for a local event e, O_e is reduced to the automaton involved in this event and that only one transition is triggered.

Fig. 1 presents an example. The first automaton $\mathcal{A}^{(1)}$ has three states $x^{(1)}$, $y^{(1)}$, and $z^{(1)}$; the second automaton $\mathcal{A}^{(2)}$ has two states $x^{(2)}$ and $y^{(2)}$. The events of this model are:

- e_1, e_2 and e_3: local events involving only $\mathcal{A}^{(1)}$, with constant rates respectively equal to λ_1, λ_2 and λ_3;
- e_4: a synchronizing event involving $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$, with a constant rate λ_4;
- e_5: a local event involving $\mathcal{A}^{(2)}$, with a functional rate f:
  - $f = \mu_1$, if $\mathcal{A}^{(1)}$ is in state $x^{(1)}$;
  - $f = 0$, if $\mathcal{A}^{(1)}$ is in state $y^{(1)}$;
  - $f = \mu_2$, if $\mathcal{A}^{(1)}$ is in state $z^{(1)}$.

When the SAN is in state ($z^{(1)}$, $y^{(2)}$), the event e_4 can occur at rate λ_4, and the resulting state of the SAN can be either ($y^{(1)}$, $x^{(2)}$) with probability π or ($x^{(1)}$, $x^{(2)}$) with probability $1 - \pi$.

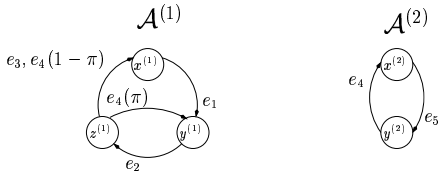

Fig. 1. Very Simple SAN model example

We see then that a SAN model is described as a set of automata (each automaton containing nodes, edges and labels). These may be used to generate the transition matrix of the Markov chain representing its dynamic behavior

using only elementary matrices. This formulation of the transition matrix is called the SAN descriptor.

2.2 PEPS 2003 textual format

A textual formalism for describing models is proposed, and it keeps the key feature of the SAN formalism: its modular specification. PEPS 2003 incorporates a graph-based approach which is close to model semantics. In this approach each automaton is represented by a graph, in which the nodes are the states and the arcs represent the occurrence of events. This textual description has been kept simple, extensible and flexible.

- Simple because there are few reserved words, just enough to delimit the different levels of modularity;
- Extensible because the definition of a SAN model is performed hierarchically;
- Flexible because of the inclusion of replication structures which allow the reuse of identical automata, and the construction of automata having repeated state blocks with the same behavior, such as found in queueing models.

This section describes the PEPS 2003 textual formalism used to describe SAN models. To be compatible with PEPS 2003, any file describing a SAN should have the suffix `.san`. Fig. 2 shows an overview of the PEPS input structure. A SAN description is composed of five blocks (Fig. 2) which are easily located with their delimiters[4] (in **bold**). The other reserved words in the PEPS input language are indicated with an *italic* font. The symbols "<" and ">" indicate mandatory information to be defined by the user. The symbols "{" and "}" indicate optional information.

The first block **identifiers** contains all declarations of parameters: numerical values, functions, or sets of indices (domains) to be used for replicas in the model definition. An identifier (< *id_name* >) can be any string of alphanumeric characters. The numerical values and functions are defined according to a C-like syntax. In general, the expressions are similar to common mathematical expressions, with logical and arithmetic operators. The arguments of these expressions can be constant input numbers (input parameters of the model), automata identifiers or state identifiers. In this last case, the expressions are functions defined on the SAN model state space. For example, "the number of automata in state $n0$" (which gives an integer result) can be expressed as "*nb* $n0$". A function that returns the value 4 if two automata ($A1$ and $A2$) are in different states, and the value 0 otherwise, is expressed as "($st\ A1\ !=\ st\ A2$) $*$ 4". Comparison operators return the value "1" for a true result and the value "0" for a false result[5].

[4] The word "delimiters" is used to indicate necessary symbols, having a fixed position in the file.

[5] In the PEPS 2003 user manual, a full description of the possible functions and the full grammatical definition of the textual format of SAN models is available [15].

```
identifiers
   < id_name >=< exp > ;
   < id_name >= [domain] ;

events
   // without replication
   loc < evt_name >< rate >< automaton >
   syn < evt_name >< rate >< automata >
   // with replication
   loc < evt_name > [domain] < rate >< automata > [domain]
   syn < evt_name > [domain] < rate >< automata > [exp − domain]

{partial} reachability =< exp > ;

network < net_name > (< type >)
aut < aut_name > {[domain]}
   stt < stt_name > {[domain]}
      to( < stt_name > )
         < evt_name > {< prob >}
   stt < stt_name >
      to( < stt_name > )
         < evt_name > {< prob >}

results
   < res_name >=< exp > ;
```

Fig. 2. Modular structure of SAN textual format

Sets of indices are useful for defining numbers of events, automata, or states that can be described as replications. A group of replicated automata of A with the set in index $[0 \ldots 2, 5, 8 \ldots 10]$ defines the set containing the automata $A[0]$, $A[1]$, $A[2]$, $A[5]$, $A[8]$, $A[9]$, and $A[10]$.

The **events** block defines each event of the model given

- its type (local or synchronizing);
- its name (an identifier);
- its firing rate (a constant or function previously defined in the identifiers block);
- its O_e set, *i.e.*, the automaton, for local events, or the automata, for synchronizing events, concerned by this event.

Additionally, events can be replicated using the sets of indexes (domains). This facility can be used when events with the same rate appear in a set of automata.

The **reachability** block contains a function defining the reachable state space of the SAN model. Usually, this is a Boolean function, returning a nonzero value for states of $\hat{S}$ that belongs to S. A model where all the states are reachable has the reachability function defined as any constant different from zero, *e.g.*, the value 1. Optionally, a partial reachability function can be defined by adding the reserved word "*partial*". In this case, only a subset of S is defined, and the overall S will be computed by PEPS 2003.

The **network** block is the major component of the SAN description and has a hierarchical structure: a network is composed of a set of automata; each automaton is composed of a set of states; each state is connected to a set of output arcs; and each arc has a set of labels identifying events (local or synchronizing)

that may trigger this transition. The first level, named "**network**", includes general information such as the name of the model "< *net_name* >" and the type of time scale of the model, namely "*continuous*" or "*discrete*". Currently, only "*continuous*" model analysis is available in PEPS 2003.

The delimiter of the automaton is the reserved word "*aut*" and "< *aut_name* >" is the automaton identifier. Optionally, a [*domain*] definition can be used to replicate it, *i.e.*, to create a number of copies of this automaton. In this case, if i is a valid index of the defined [*domain*] and A the name of the replicated automaton, $A[i]$ is the identifier of one automaton.

The *stt* section defines a local state of the automaton. "< *stt_name* >" is the state identifier, and [*domain*] can be used to create replicas of the state. A description of each output transition from this state is given by the definition of a "*to()*" section. The identifier "< *stt_name* >" inside the parenthesis indicates the output state of this transition. Inside a group of replicated states, the expression of the other states inside the group can be made by positional references to the current (==), the previous (– –) or the successor (++). Larger jumps, *e.g.*, of states two ahead (+2), can also be defined, but any positional reference pointing to a non-existing state or to a state outside the replicated group is ignored. Finally, for each transition defined, a set of events (local and synchronizing) that can trigger the transition can be expressed by their names ("< *evt_name* >") and optionally (if different from 1) the probability of occurrence. The *from* section is quite similar to the *stt* section, but it cannot define local states. This is commonly used to define additional transitions which cannot be defined in the *stt* section. A typical use of the *from* section is to define a transition leaving from only one state of a group of replicated states to a state outside the group, *e.g.*, a queue with particular initial or final states may need this kind of transition definition.

The functions used to compute performance indexes of the SAN model are defined in the **results** block. The results obtained by PEPS are the mean values of these functions computed using the stationary probability solution of the model.

3 The PEPS 2003 Software Tool

PEPS is implemented using the `C++` programming language, and although the source code is quite standard, only Linux and Solaris version have been tested. The main features of version 2000 are [7]:

- Textual description of continuous-time SAN models (without replicas);
- Stationary solution of models using Arnoldi, GMRES and Power iterative methods [16, 19];
- Numerical optimization regarding functional dependencies, diagonal precomputation, preconditioning and algebraic aggregation of automata [6]; and
- Results evaluation.

PEPS 2003 includes some bug corrections and three new features:

- Compact textual description of continuous-time SAN models;

- Numerical solution using probability vectors with the size of the reachable state space; and
- Fast(er) function evaluation.

The previous Section presented the new textual format used in PEPS 2003 to allow compact descriptions mostly due to the idea of replications of automata, states and events. The next two sections (3.1 and 3.2) present the other new features of version 2003. This paper will not present details on how to install and operate PEPS; to learn how to do so, the reader is invited to read the user manual available at the PEPS home page [15].

3.1 Probability Vector Handling

SAN allow Markov chain models to be described in a memory efficient manner because their storage is based on a tensor structure (descriptor). However, the use of independent components connected via synchronizations and functions may produce a representation with many unreachable states ($|S| \ll |\hat{S}|$). Within this tensor (Kronecker) framework, a number of algorithms have been proposed to compute the product of a probability vector and the descriptor. The first and perhaps best-known, is the shuffle algorithm [3, 6, 7], which computes the product but never needs the matrix explicitly. However, this algorithm needs to use "extended" vectors $\hat{\pi}$ whose size is equal to that of $\hat{S}$. This algorithm is denoted **E-Sh**, for **extended shuffle**. This algorithm was the only one available in PEPS 2000.

However, when there are many unreachable states ($|S| \ll |\hat{S}|$), E-Sh is not efficient, because of its use of extended vectors. The probability vector will have many zero elements, since only states corresponding to reachable states have nonzero probability. Moreover, computations are carried out for all the elements of the vector, even those elements corresponding to unreachable states. Therefore, the computation gain obtained by exploiting the tensor formalism may be negated if many useless computations are performed. Furthermore, memory is used for states whose probability is always zero.

The use of *reduced* vectors (vectors π which contains entries only for reachable states, *i.e.*, vectors of size $|S|$) allows a reduction in memory needs, and some unneeded computations are avoided. This leads to significant memory gains when using iterative methods such as Arnoldi or GMRES which can possibly require many probability vectors. A modification to the E-Sh shuffle algorithm permits the use of such vectors. However, to obtain good performance at the computation-time level, some intermediate vectors of size $\hat{S}$ are also used. An algorithm described in [3] and implemented in PEPS 2003 allows us to save computations by taking into account the fact that probabilities corresponding to non-reachable states are always zero in the resulting vector. This **partially reduced** computation corresponds to the algorithm called **PR-Sh**. However, the savings in memory turns out to be somewhat insignificant for the shuffle algorithm itself.

A final version of the shuffle algorithm concentrates on the amount of memory used, and allows us to handle even more complex models. In this algorithm, all the intermediate data structures are stored in reduced format. This **fully reduced** computation corresponds to the algorithm called **FR-Sh**. This algorithm is also implemented in PEPS 2003, and described in [3].

3.2 Fast Function Evaluation

One of the most important improvements of the new version of the PEPS software is the inclusion of fast function evaluation. The use of functions is one of the key features of PEPS . It is an intuitive and compact form to represent interdependences among automata. An efficient numerical solution of SAN models with functions was published in [7]. This work described properties of the tensor operations and algorithms to reduce the number of function evaluations performed during vector multiplications.

In the previous version of PEPS (version 2000), the functions were represented in reverse polish notation which is interpreted during execution time and evaluated using a stack implemented as a regular data structure. This implementation is quite flexible, and it allows the transformation of functions during the compilation of a SAN model[6]. However, the interpretation of functions may cost a great amount of time during execution. Typically, for a functional rate in automata i, denoting n_j the number of local states of the j-th automaton, and N the number of automata, each functional element in a matrix corresponding to the i-th automaton can be evaluated up to $(\prod_{j=1,j\neq i}^{N} n_j)$ times. Additionally, since PEPS solves the models using iterative methods, all these evaluations must be carried out during each and every iteration. The efficient algorithm presented in [7] can take advantage of the particular dependencies of the functional elements to reduce this number of evaluations, but very often this number remains quite large.

The natural alternative to function interpretation is to compile the functions to generate executable code. Since each SAN model has its own functions, the functional elements are compiled into `C++` code by a system call to the *gcc* compiler to generate a dynamic library which is called by the PEPS software. Such a technique is similar to the *just in time* code generation technique employed in Java environments [20]. In the case of PEPS however, the purpose of such real time code generation is not to provide machine independence, but to provide a tailor-made function evaluation to each SAN model.

[6] The functions defined in the section `identifiers` of the model textual description are rarely used as stated. The generation of the internal representation of a model, *i.e.*, the operand matrices of the Markovian descriptor, usually create new functions. This is done, *e.g.*, for the normalization of the descriptor, where all matrix elements must be divided by a normalizing factor. In the case of automata algebraic grouping, even more new functions must be created due to the symbolic evaluation of matrix elements that should be added (in the tensor sum local part) or multiplied (in the tensor product synchronized part).

The gains obtained by the new function evaluation are substantial during the iterative solution of the SAN models, but the flexibility of the interpreted evaluation still justifies its use during the compilation of the model and the normalization of the descriptor. A version of PEPS 2003 without the compiled function evaluation is also available to run PEPS on platforms where the *gcc* compiler is not available.

4 Examples of SAN models in PEPS 2003

In this section three examples are presented to illustrate the modeling power and the computational effectiveness of PEPS 2003 . For each example, the generic SAN model is described and then numerical results are computed for some specific cases. The machine used to run the examples is an IBM-PC running Linux OS (Mandrake distribution, v.8.0, with 1.5 Gbytes of RAM and with 2.0 GHz Pentium IV processor. PEPS 2003 was compiled using gcc with the optimization option `-O3`. The indicated processing times do not take system time into account, *i.e.*, they refer only to the user time spent for one iteration.

4.1 Example: A Model of Resource Sharing

The first example is a traditional resource sharing model, where N distinguishable processes share a certain amount (R) of indistinguishable resources. Each of these processes alternates between a *sleeping* and a resource *using* state. When a process wishing to move from the sleeping to the using state finds R processes already using the resources, that process fails to access the resource and it returns to the sleeping state. Notice that when $R = 1$ this model reduces to the usual mutual exclusion problem. Analogously, when $R = N$ all the processes are independent and there is no restriction to access the resources. We shall let λ_i be the rate at which process i awakes from the sleeping state wishing to access the resource, and μ_i, the rate at which this same process releases the resource.

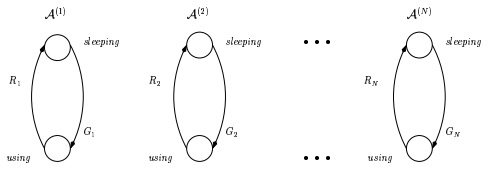

Fig. 3. Resource Sharing Model - version 1

In our SAN representation (Fig. 3), each process is modeled by a two state automaton $\mathcal{A}^{(i)}$, the two states being *sleeping* and *using*. We shall let $st\mathcal{A}^{(i)}$ denote the current state of automaton $\mathcal{A}^{(i)}$. Also, we introduce the function

$$f = \delta\left(\sum_{i=1}^{N} \delta(st\mathcal{A}^{(i)} = using) < R\right).$$

where $\delta(b)$ is an integer function that has the value 1 if the Boolean b is true, and the value 0 otherwise. Thus the function f has the value 1 when access to the resource is permitted and has the value 0 otherwise. Fig. 3 provides a graphical illustration of this model, called RS1. In this representation each automaton $\mathcal{A}^{(i)}$ has two local events:

- G_i which corresponds to the i-th process *getting* a resource, with rate $\lambda_i\ f$;
- R_i which corresponds to the i-th process *releasing* a resource, with rate μ_i.

The textual `.san` file describing this model is:

```
//==================== RS model version 1 ==================
//                          N=4, R=2
//==========================================================
identifiers
   R       = 2;   // amount of resources
   mu1     = 6;   // rate for leaving a resource for process 1
   lambda1 = 3;   // rate for requesting a resource for process 1
   f1      = lambda1 * (nb using < R);
   mu2     = 5;   // rate for leaving a resource for process 2
   lambda2 = 4;   // rate for requesting a resource for process 2
   f2      = lambda2 * (nb using < R);
   mu3     = 4;   // rate for leaving a resource for process 3
   lambda3 = 6;   // rate for requesting a resource for process 3
   f3      = lambda3 * (nb using < R);
   mu4     = 3;   // rate for leaving a resource for process 4
   lambda4 = 5;   // rate for requesting a resource for process 4
   f4      = lambda4 * (nb using < R);

events
   loc G1 (f1)  P1  // local event G1 has rate f1 and appears in automaton P1
   loc R1 (mu1) P1  // local event R1 has rate mu1 and appears in automaton P1
   loc G2 (f2)  P2  // local event G2 has rate f2 and appears in automaton P2
   loc R2 (mu2) P2  // local event R2 has rate mu2 and appears in automaton P2
   loc G3 (f3)  P3  // local event G3 has rate f3 and appears in automaton P3
   loc R3 (mu3) P3  // local event R3 has rate mu3 and appears in automaton P3
   loc G4 (f4)  P4  // local event G4 has rate f4 and appears in automaton P4
   loc R4 (mu4) P4  // local event R4 has rate mu4 and appears in automaton P4

reachability = (nb using <= R);     // only the states where at the most R
                                    // resources are being used are reachable
network rs1 (continuous)
  aut P1 stt sleeping to(using)    G1
         stt using    to(sleeping) R1
  aut P2 stt sleeping to(using)    G2
         stt using    to(sleeping) R2
  aut P3 stt sleeping to(using)    G3
         stt using    to(sleeping) R3
  aut P4 stt sleeping to(using)    G4
         stt using    to(sleeping) R4

results
  full     = nb using == R;     // probability of all resources being used
  empty    = nb using == 0;     // probability of all resources being available
  use1     = st P1 == using;    // probability that the first process uses the resource
  average  = nb using;          // average number of occupied resources
```

It was not possible to use replicators to define all four automata in this example. In fact, the use of replications is only possible if all automata are identical, which is not the case here since each automaton has different events (with different rates). If all the processes had the same acquiring (λ) and releasing (μ) rates, this example could be represented more simply as:

```
//========== RS model version 1 with same rates =============
//                          N=4, R=2
//============================================================
identifiers
   N       = [1..4];  // amount (and identifier) of processes
   R       = 2;       // amount of resources
   mu      = 6;       // rate for leaving a resource for all processes
   lambda  = 3;       // rate for requesting a resource for all processess
   f       = lambda * (nb using < R);

events
   loc G  (f)   P  // local event G has rate f and appears in all automata P
   loc R  (mu)  P  // local event R1 has rate mu and appears in all automata P

reachability = (nb using <= R);     // only the states where at the most R
                                    // resources are being used are reachable
network rs1 (continuous)
  aut P[proc] stt sleeping to(using)    G
              stt using    to(sleeping) R

results
  full     = nb using == R;     // probability of all resources being used
  empty    = nb using == 0;     // probability of all resources being available
  use1     = st P[1] == using;  // probability that the first process uses the resource
  average  = nb using;          // average number of occupied resources
```

We wish to point out that in the PEPS documentation, a number of variants of this model are included, to show that it is possible with only simple modifications to introduce a complete set of related models. Within the scope of this paper, it is interesting to describe a specific variation of this model that describes exactly the same problem, but which does not use functions to represent the resource contention. In fact, in this case, and in many others, synchronizing events can be used to generate an equivalent model without functional rates (freqeuntly with many more automata, states, and/or synchronizing events). Fig. 4 presents this new model, where an automaton is introduced to represent the resource pool. The resource allocation (events G_i, rate λ_i) and release (events R_i, rate μ_i) that were formerly described as local events, will now be synchronizing events that increment the number of occupied resources at each possible allocation, and decrement it at each release. The resource contention is modeled by the impossibility of a process passing to the using state when all resources are occupied, *i.e.*, when the automaton representing the resource is in the last state (where only release events can happen).

The PEPS 2003 textual format of this model is as follows:

```
//==================== RS model version 2 ==================
//                          N=4, R=2
//============================================================
identifiers
   R        = 2;       // amount of resources
   mu1      = 6;       // rate for leaving a resource for process 1
   lambda1  = 3;       // rate for requesting a resource for process 1
   mu2      = 5;       // rate for leaving a resource for process 2
   lambda2  = 4;       // rate for requesting a resource for process 2
   mu3      = 4;       // rate for leaving a resource for process 3
   lambda3  = 6;       // rate for requesting a resource for process 3
   mu4      = 3;       // rate for leaving a resource for process 4
   lambda4  = 5;       // rate for requesting a resource for process 4
   res_pool = [0..R];  // domain to describe the available resources pool
```

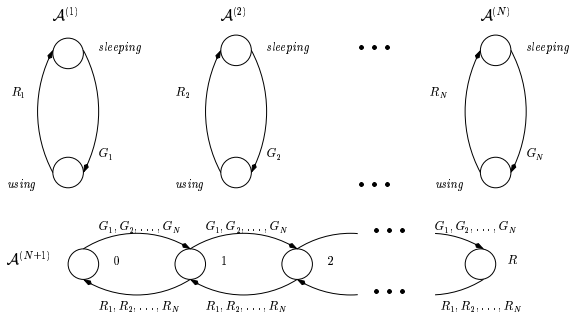

Fig. 4. Resource Sharing Model without functions - version 2

```
events
   syn G1 (f1)  P1 RP  // event G1 has rate f1 and appears in automata P1 and RP
   syn R1 (mu1) P1 RP  // event R1 has rate mu1 and appears in automata P1 and RP
   syn G2 (f2)  P2 RP  // event G2 has rate f2 and appears in automata P2 and RP
   syn R2 (mu2) P2 RP  // event R2 has rate mu2 and appears in automata P2 and RP
   syn G3 (f3)  P3 RP  // event G3 has rate f3 and appears in automata P3 and RP
   syn R3 (mu3) P3 RP  // event R3 has rate mu3 and appears in automata P3 and RP
   syn G4 (f4)  P4 RP  // event G4 has rate f4 and appears in automata P4 and RP
   syn R4 (mu4) P4 RP  // event R4 has rate mu4 and appears in automata P4 and RP

reachability = (nb [P1..P4] using == st RP);
   // the number of Processes using ressources must be equal to number
   //  of occupied ressources in the Ressource Pool

network rs2 (continuous)
  aut P1 stt sleeping to(using)    G1
         stt using    to(sleeping) R1
  aut P2 stt sleeping to(using)    G2
         stt using    to(sleeping) R2
  aut P3 stt sleeping to(using)    G3
         stt using    to(sleeping) R3
  aut P4 stt sleeping to(using)    G4
         stt using    to(sleeping) R4
  aut RP stt n[res_pool]
           to(++) G1 G2 G3 G4
           to(--) R1 R2 R3 R4

results
  full     = st RP == n[R];     // probability of all resources being used
  empty    = st RP == n[0];     // probability of all resources being available
  use1     = st P1 == using;    // probability of the first process use the resource
  average  = st RP;             // average number of occupied resources
```

4.2 Example: First Server Available Queue

The second example considers a queue with common exponential arrival and a finite number (C) of distinguishable and ordered servers ($C_i, i = 1 \ldots C$). As a client arrives, it is served by the first available server, *i.e.*, if C_1 is available, the client is served by it, otherwise if C_2 is available the client is served by it, and so on. This queue behavior is not monotonic, so, as far as we can ascertain, there is no product-form solution for this model. The basic technique to model this queue is to consider each server as a two-state automaton (states *idle* and *busy*). The arrival in each server is expressed by a local event (called L_i) with a

functional rate that is nonzero and equal to λ, if all preceding servers are busy, and zero otherwise. At a given moment, only one server, the first available, has a nonzero arrival rate. The end of service at each server is simply a local event (D_i) with constant rate μ_i.

The same model can also be expressed as a SAN without functions. In this case, each function is replaced by a synchronizing event that synchronizes the automaton representing the server accepting a client with all previous automata in the busy state. The PEPS 2003 textual formats for the original (with functions) and this alternative model are as follows:

```
//================== FSA model ==============
//         (with functions)   C=4
//==========================================
identifiers
   lambda  = 5;
   mu1     = 6;
   f1      = lambda;
   mu2     = 5;
   f2      = (st C1 == busy) * lambda;
   mu3     = 4;
   f3      = (nb[C1..C2] busy == 2) * lambda;
   mu4     = 3;
   f4      = (nb[C1..C3] busy == 3) * lambda;
events
   loc L1 (f1)  C1
   loc D1 (mu1) C1
   loc L2 (f2)  C2
   loc D2 (mu2) C2
   loc L3 (f3)  C3
   loc D3 (mu3) C3
   loc L4 (f4)  C4
   loc D4 (mu4) C4

reachability = 1;

network fsa (continuous)
  aut C1 stt idle to(busy) L1
         stt busy to(idle) D1
  aut C2 stt idle to(busy) L2
         stt busy to(idle) D2
  aut C3 stt idle to(busy) L3
         stt busy to(idle) D3
  aut C4 stt idle to(busy) L4
         stt busy to(idle) D4
results
  full     = nb busy == C;
  empty    = nb busy == 0;
  use1     = st P1 == busy;
  average  = nb busy;
```

```
//============ FSA alternative model =======
//     (with synchronizing events)    C=4
//==========================================
identifiers
   lambda  = 5;
   mu1     = 6;
   mu2     = 5;
   mu3     = 4;
   mu4     = 3;
events
   loc L1 (lambda)  C1
   loc D1 (mu1)     C1
   syn L2 (lambda)  C1 C2
   loc D2 (mu2)     C1
   syn L3 (lambda)  C1 C2 C3
   loc D3 (mu3)     C3
   syn L4 (lambda)  C1 C2 C3 C4
   loc D4 (mu4)     C4

reachability = 1;

network fsa2 (continuous)
   aut C1 stt idle to(busy) L1
          stt busy to(idle) D1
                   to(busy) L2 L3 L4
   aut C2 stt idle to(busy) L2
          stt busy to(idle) D2
                   to(busy) L3 L4
   aut C3 stt idle to(busy) L3
          stt busy to(idle) D3
                   to(busy) L4
   aut C4 stt idle to(busy) L4
          stt busy to(idle) D4

results
   full     = nb busy == C;
   empty    = nb busy == 0;
   use1     = st P1 == busy;
   average  = nb busy;
```

4.3 Example: A Mixed Queueing Network

The final example is a mixed queueing network (Fig. 5) in which customers of class 1 arrive to and eventually depart (i.e., open) and customers of class 2 circulate forever in the network, (i.e., closed). This quite complex example is presented to stress the power of description of PEPS 2003 , and to provide a really large SAN model. Due to its size, the equivalent SAN model is not presented as a figure. However, the construction technique does not differ significantly from the technique employed with the previous models. In this model, each

queue visited by only the first class of customer (Queues 2 and 3) is represented by one automaton each ($\mathcal{A}^{(2^1)}$ and $\mathcal{A}^{(3^1)}$, respectively). Queues visited by two classes of customers are represented by two automata (one for each class) and the total number of customers in a queue is the sum of customers (of both classes) represented in each automaton. The size of this model depends on the maximum capacity of each queue, denoted K_i for queue i. For the second class of customer (closed system) it is also necessary to define the number of customers in the system (N_2). In this example, all queues block when the destination queue is full, even though other behavior, *e.g.*, loss, could be easily modeled with the SAN formalism.

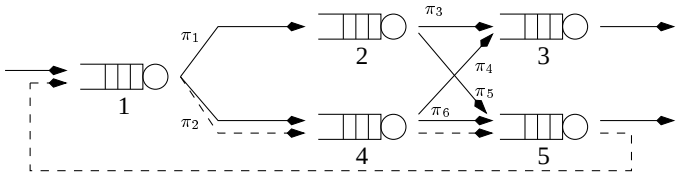

Fig. 5. Mixed queueing network

The equivalent SAN model for this example has eight automata ($\mathcal{A}^{(1^1)}$, $\mathcal{A}^{(1^2)}$, $\mathcal{A}^{(2^1)}$, $\mathcal{A}^{(3^1)}$, $\mathcal{A}^{(4^1)}$, $\mathcal{A}^{(4^2)}$, $\mathcal{A}^{(5^1)}$, $\mathcal{A}^{(5^2)}$) representing each possible pair (customer, queue). The model has two local events (arrival and departure of class 1 customers), and nine synchronizing events (the routing paths for customers from both classes). Functional transitions are used to represent the capacity restriction of admission in queues accepting both classes of customer. The reachability function of the SAN model representing this queueing network must take into account both the unreachable states due to the use of two automata to represent a queue accepting two classes of customer and the fixed number of customers of class 2.

Assuming queue capacities $K_1 = 10$, $K_2 = 5$, $K_3 = 5$, $K_4 = 8$, $K_5 = 8$, and a total population of class 2 customers, N_2, equal to 10, the reachability function for this model is:

$$
\begin{aligned}
reachability = & \left[\left(st(\mathcal{A}^{(1^1)}) + st(\mathcal{A}^{(1^2)})\right) \leq 10\right] \ and \\
& \left[\left(st(\mathcal{A}^{(4^1)}) + st(\mathcal{A}^{(4^2)})\right) \leq 8\right] \ and \\
& \left[\left(st(\mathcal{A}^{(5^1)}) + st(\mathcal{A}^{(5^2)})\right) \leq 8\right] \ and \\
& \left[\left(st(\mathcal{A}^{(1^2)}) + st(\mathcal{A}^{(4^2)}) + st(\mathcal{A}^{(5^2)})\right) == 10\right]
\end{aligned}
$$

The equivalent SAN model has a product state space containing $28,579,716$ states of which only $402,732$ are reachable. The complete description and `.san` file for this model can be obtained from the PEPS web page [15].

5 Numerical results

Table 1 shows some numerical results obtained by PEPS 2003. The different columns indicates the model, its product state space and reachable state space sizes, the time for a single vector-descriptor multiplication with PEPS 2000 and with PEPS 2003 with the extended and the fully reduced shuffle algorithms.

Table 1. PEPS 2003 numerical results

Models	*pss*	*rss*	*1 iteration time*		
			PEPS2000	PEPS2003 - E-Sh	PEPS2003 - FR-Sh
RS with functions N=20, R=1	1,048,576	21	23.1	14.5	1.5
RS with functions N=20, R=5	1,048,576	21,700	23.1	14.5	2.0
RS with functions N=20, R=10	1,048,576	616,666	23.1	14.5	15.8
RS with functions N=20, R=15	1,048,576	1,042,380	23.2	14.6	28.0
RS with functions N=20, R=19	1,048,576	1,048,575	23.2	14.6	29.2
RS with functions N=20, R=20	1,048,576	1,048,576	1.0	1.1	17.7
RS with synch. events N=20, R=1	2,097,152	21	4.8	7.7	4.6
RS with synch. events N=20, R=5	6,291,456	21,700	22.2	22.8	7.1
RS with synch. events N=20, R=10	11,534,336	616,666	43.3	41.0	28.8
RS with synch. events N=20, R=15	16,777,216	1,042,380	61.1	60.7	52.9
RS with synch. events N=20, R=19	20.971.520	1,048,575	76.0	74.7	59.3
RS with synch. events N=20, R=20	22.020.096	1,048,576	77.4	76.7	59.6
FSA with functions C=25	33,554,432	33,554,432	245.2	154.9	-
FSA with synch. events C=25	33,554,432	33,554,432	339.1	328.4	-
Mixed Queueing Network	28,579,716	402,732	79.1	52.8	71.4

Examining the results of the "resource sharing" model with functions[7], the values obtained for PEPS 2000 show that the cost of function evaluation is high (when $N = R$ PEPS automatically removes the functions). The small variations come from the fact that the number of function evaluations changes according to R. This cost is decreased by a factor of 2/3 when using compiled functions in PEPS 2003. The use of the **FR-Sh** algorithm gives better times when the size of the reachable state space is less than 50% of the product state space. The curve plotted in the first graphic of Fig. 6 shows this more clearly.

The results of the resource sharing model with synchronizing events display similar results for PEPS 2000 and PEPS 2003, as there are no functions in the

[7] No distinctions between the RS1 model with the same or different rates is made since their numerical results are identical.

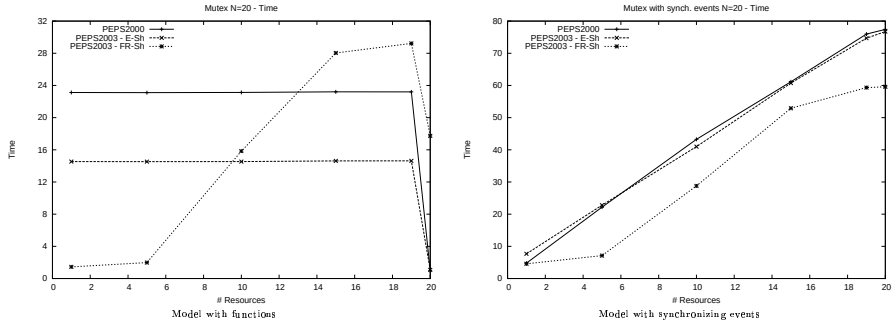

Fig. 6. Computational time for RS models

model. On the other hand, the use of the **FR-Sh** algorithm is beneficial when the ratio between the reachable state space and the product state space is high as demonstrated by the curve in the second graphic of Fig. 6.

For the "first available server" example, the reachable state space is equal to the product state space, so the **FR-Sh** algorithm does not bring any benefit. The model with functions has better times in PEPS 2000 than the model without functions, because, even with the same state spaces, the descriptor is more complex with synchronizing events than with functions. Moreover, in the model with function, PEPS 2003 provides better performance. This is a case in which the benefits of using functional rates is clear.

For the "mixed queueing network" example, a curious phenomenon appears. The rather good ratio (less than 2%) between product and reachable state space should suggest a much better performance of the sparse vector techniques. However, the complexity of the SAN Markovian descriptor of this model seems to be unsuitable. In this model almost all the transition rates are associated with synchronizing events and very few are associated with local events. The FR-Sh algorithm includes an overhead for descriptor parts that refer to synchronizing events, because intermediate computation vectors may leave the reachable state space. In any case, a more complete study on the benefits, and possibly some improvements, of **FR-Sh** sparse vectors algorithms applied to such models is a natural for future work. This is now facilitated by the availability of PEPS 2003.

A natural extension of this work includes studies aimed at finding the best techniques for each class of problem modeled by SAN. Also, the evolution of interest in structural representations and the rapid evolution of numerically efficient methods also suggest further versions of PEPS. The authors current work will provide, in the near future, a new version of PEPS which will include new algorithms to perform automatic lumping of models with replicas and which will offer simulation algorithms to solve SAN models.

References

1. M.Ajmone-Marsan, G.Balbo, G.Conte, S.Donatelli, G.Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* John-Wiley, 1995.
2. K. Atif and B. Plateau. Stochatic Automata Networks for Modeling Parallel Systems. *IEEE Transactions on Software Engineering*, v.17, n.10, pp.1093-1108, 1991.
3. A. Benoit, B. Plateau and W.J. Stewart. Memory-efficient Kronecker algorithms with applications to the modelling of parallel systems. *To appear in PMEO-PDS'03*, 2003.
4. G.F.Ciardo, A.S.Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In: *Proc. 8th International Workshop on Petri Nets and Performance Evaluation*, 1999.
5. S.Donnatelli. Superposed Stochastic Automata: a Class of Stochastic Petri Nets with Parallel Solution and Distributed State Space. *Performance Evaluation*, v.18, pp.21-36, 1993.
6. P.Fernandes. *Méthodes Numériques pour la Solution de Systèmes Markoviens à Grand Espace d'Etats.* Thèse de doctorat, Institut National Polytechnique de Grenoble, France, 1998.
7. P.Fernandes, B.Plateau and W.J.Stewart. Efficient Descriptor-Vector Multiplication in Stochastic Automata Networks. *Journal of the ACM*, v.45, n.3, pp.381-414, 1998.
8. J.Fourneau, B.Plateau. A Methodology for Solving Markov Models of Parallel Systems. *Journal of Parallel and Distributed Computing*, v.12, pp.370-387, 1991.
9. E.Gelenbe, G.Pujolle. *Introduction to Queueing Networks.* John Wiley, 1997.
10. J.Hillston. *A Compositional Approach for Performance Modelling.* Ph.D. Thesis, University of Edinburg, United Kingdom, 1994.
11. J.K.Muppala, G.F.Ciardo, K.S.Trivedi. Stochastic Reward Nets for Reliability Prediction. *Communications in Reliability, Maintainability and Serviceability*, v.1, n.2, pp.9-20, 1994.
12. B.Plateau. *De l'Evaluation du Parellélisme et de la Synchronisation.* Thèse de Doctorat d'Etat, Paris-Sud, Orsay, France, 1984.
13. B.Plateau. On the Stochastic Structure of Parallelism and Synchronization Models for Distributed Algorithms. In: *Proc. ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, Austin, Texas, 1985.
14. B.Plateau, J.M.Fourneau, K.Lee. PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In: R.Puigjaner, D.Potier, eds. *Modelling Techniques and Tools for Computer Performance Evaluation*, 1988.
15. PEPS team. PEPS 2003 *Software Tool.* On-line document available at `http://www-apache.imag.fr/software/peps`, visited Feb. 14th, 2003.
16. Y.Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1995.
17. W.H.Sanders, J.F.Meyer. An Unified Approach for Specifying Measures of Performance, Dependability, and Performability. *Dependable Computing for Critical Applications*, v.4, pp.215-238, 1991.
18. W.J.Stewart. MARCA: Markov Chain Analyzer. *IEEE Computer Repository* No. R76 232, 1976.
19. W.J.Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.
20. Sun Microsystems *The JIT Compiler Interface Specification.* On-line document available at `http://java.sun.com/docs/jit_interface.html`, visited Feb. 14th, 2003.