PEPS 2003 User manual

The PEPS team
Anne Benoit, Leonardo Brenner, Paulo Fernandes,
Brigitte Plateau, William J. Stewart

October 6, 2003

Abstract

PEPS is a software which allows to solve numerically very large Markov Chains. It uses an input interface based on Stochastic Automata Networks (SAN), a compact storage for the transition matrix of the Markov Chain and tensor algebra to handle the basic vector matrix multiplication. Solution methods are iterative.

Key Features

- Provides a powerful and modular interface, the SAN
- Generates automatically very large Markov Chain transition matrices
- Allows a very compact storage and can generate sparse format (Marca compatible)
- Offers various numerical strategies to improve the time/space trade-off (grouping and lumping)
- Includes several numerical iterative methods, with or without preconditionning
- Computes performances indexes from steady-state probability vector

Application Area

- telecommunications
- computer performance modeling
- distributed and parallel systems
- software engineering
- finite queueing networks (with blocking, priorities, contention, etc.)

Key words: Markov Chains, Stochastic Automata Networks, Generalized tensor algebra, Vector-descriptor multiplication.

Contents

1 Introduction **1**

1.1 SAN Presentation 2

1.2 PEPS Presentation 3

2 SAN **Textual Interface** **5**

2.1 Identifiers and Domains 5

2.2 Events 7

2.3 Reachability Function 8

2.4 Network Description 8

2.4.1 Automaton Description 9

2.4.2 State Description 9

2.4.3 Transition Description 10

2.4.4 Event Description 10

2.5 Results Description 11

2.6 Mathematical Expressions 11

3 Modeling Examples **15**

3.1 A Model of Resource Sharing 15

3.2 First Server Available Queue 19

3.3 Example: A Mixed Queueing Network 21

4 Using PEPS 2003 **24**

4.1 PEPS 2003 Installation 24

4.1.1 Getting the source 24

4.1.2 Compiling 24

4.1.3 Binary files 24

4.2 PEPS 2003 User Interface 25

4.2.1 Compiling a SAN model . . . 25
4.2.2 Solving a SAN model . . . 28
4.2.3 Changing preferences . . . 30
4.2.4 Probability vector facilities . . . 31
4.2.5 Loading data structures . . . 36
4.2.6 Inspect data structures . . . 36
4.2.7 Sparse matrix facilities . . . 38
4.2.8 About this version . . . 39
4.2.9 Debugging models and PEPS . . . 39
4.2.10 Errors . . . 40

Chapter 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) [7, 13] 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 indexes. The main difficulties in this process are 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 [10], generalized stochastic Petri nets [1], stochastic reward nets [12] and stochastic activity nets [18] are, thanks to their extensive modeling capabilities, widely used in diverse application domains, and notably in the areas of parallel and distributed systems.

The pioneer work that 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 [6] and process algebras [11].

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 [5], 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 [8]. 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.

1.1 SAN Presentation

In a SAN [8, 14] 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].

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 1.0.

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 an 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 $\mathcal{S}^{(i)}$, and the cardinality of $\mathcal{S}^{(i)}$ is denoted by n_i. A global state for the SAN model is a vector $\tilde{x} = (x^{(1)}, \dots, x^{(N)})$. $\mathcal{S} = \mathcal{S}^{(1)} \times \cdots \times \mathcal{S}^{(N)}$ 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 $\mathcal{R}$; it is generally smaller than the product state space since synchronizing events and functional rates may prevent some states in $\mathcal{S}$ 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.

Figure 1.1 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 $1 - \pi$.

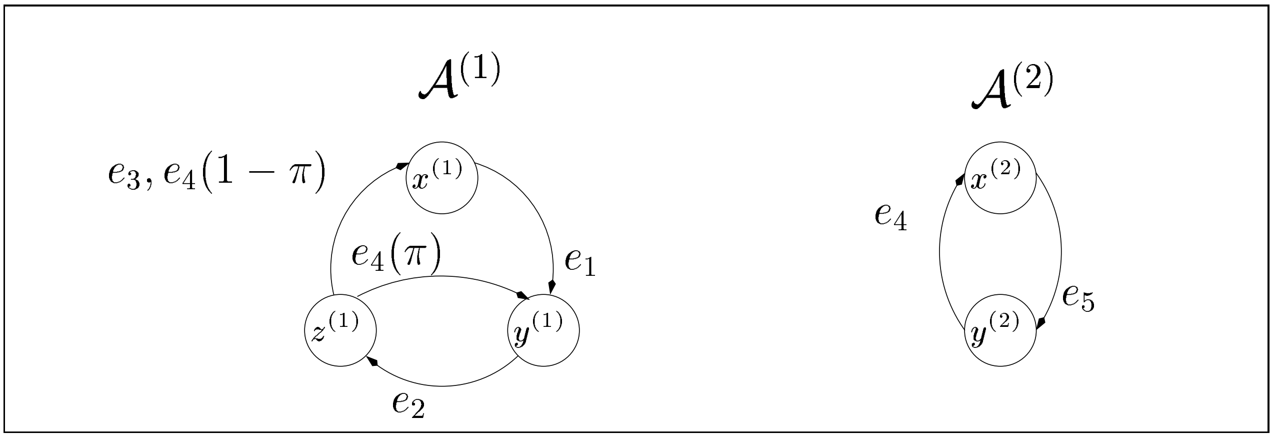

Figure 1.1: Very Simple SAN model example

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.

1.2 Peps Presentation

Peps is software package 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 [8]:

- Textual description of continuous-time SAN models (without replicas);
- Stationary solution of models using Arnoldi, GMRES and Power iterative methods [17, 20];
- Numerical optimization regarding functional dependencies, diagonal pre-computation, pre-conditioning and algebraic aggregation of automata [7]; and
- Results evaluation.

Peps 2003 includes some bug corrections and three new features:

- Compact textual description of continuous-time SAN models;
- Semantic aggregation of SAN with replicas;
- Numerical solution using probability vectors with the size of the reachable state space; and
- Fast(er) function evaluation.

PEPS 2003 is able to:

- Read a file which describes a SAN in a textual format. This interface is described in Chapter 2. Chapter 3 provides simple examples of models that can be used to experiment with the software. The SAN textual format permits a user to specify performance indexes.
- Compile a SAN model to form all the small matrices then to assemble them to construct the complete SAN descriptor. At this level, a choice of granularity is given to the user: some automata can be grouped to form an equivalent SAN with fewer, but larger automata.
- Provide the user with a selection of iterative solution methods and a number of numerical options, as it is explained in Chapter 4.
- Inspect data structures such as the reachable state space, the stationary probability vector (and compare two of them), and the SAN descriptor. See Chapter 4.
- Output results and data structure files: results, probability vectors, SAN descriptor, HBF format of the descriptor compatible with the software MARCA [19] (www.csc.ncsu.edu/faculty/WStewart/MARCA/marca.html).

Chapter 2

SAN Textual Interface

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 transitions fired by 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 the inclusion of replication structures allows 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`. Figure 2.1 shows an overview of the PEPS input structure. A SAN description is composed of five blocks (Figure 2.1) which are easily located with their delimiters[1] (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.

2.1 Identifiers and Domains

This first block **identifiers** contains all declarations of parameters: numerical values, functions, or sets of indexes (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

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

```
identifiers
  < id_name >=< exp > ;
  < dom_name >= [i..j] ;

events
  // without replication
  loc < evt_name >< rate > {automaton}
  syn < evt_name >< rate > {automata}
  // with replication
  loc < evt_name > [number_of_replications] < rate > {automata[replica]}
  syn < evt_name > [number_of_replications] < rate > {automata[exp − replica]}

{partial} reachability =< exp > ;

network < net_name > (< type >)
  aut < aut_name > {[number_of_replications]}
    stt < stt_name > {[number_of_replications]} {(reward)}
      to( < stt_name > )
        < evt_name > {(< prob >)}
        ...
        < evt_name > {(< prob >)}
      ...
    from < stt_name >
      to( < stt_name > )
        < evt_name > {(< prob >)}
    ...
    stt < stt_name > {[number_of_replications]} {(reward)}
      to( < stt_name > )
        < evt_name > {(< prob >)}
  ...
  aut < aut_name > {[number_of_replications]}
    ...

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

Figure 2.1: Modular structure of SAN textual format

"*nb n*0". 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 "($st\ A1 \mathrel{!=} st\ A2$) $*$ 4". Comparison operators return the value "1" for a true result and the value "0" for a false result. The format of an expression is described in Section 2.6.

Format of the **identifiers** block

```
identifiers
  < id_name >=< exp > ;
  ...
  < dom_name >= [i..j] ;
  ...
```

- "$<$ *id_name* $>$" is an expression identifier which begins with a letter and is followed by a sequence of letter or numbers. The maximum length of an expression identifier is 128 characters;
- "$<$ *dom_name* $>$" is a domain identifier. It is a set of indexes. A domain can be defined by an interval "[1..3]", by a list "[1,2,3]" or by list of intervals "[1..3,5,7..9]";

- "$< exp >$" is a real number or a mathematical expression. The mathematical expressions are described in Section 2.6. A real number has one of the following formats:

  an integer, such as "12345";

  a floating point real number, such as "12345.6789";

  a real number with mantissa and exponent, such as "12345E(or e)+(or –)10" or "12345.6789e+100'.

Sets of indexes 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 $[0..2, 5, 8..10]$ defines the set containing the automata $A[0]$, $A[1]$, $A[2]$, $A[5]$, $A[8]$, $A[9]$, and $A[10]$.

2.2 Events

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. This set can be omitted.

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.

Format of the **events** block
events $loc < evt\_name > [number\_of\_replications] < rate > \{automata[replica]\}$ $syn < evt\_name > [number\_of\_replications] < rate > \{automata[exp - replica]\}$ ...

- "*loc*" define the event type as local event;
- "*syn*" define the event type as synchronized event;
- "$< evt\_name >$" the event identifier begins with a letter and is followed by a sequence of letter or numbers. The maximum length of an identifier is 128 characters;
- "$[number\_of\_replications]$" is a set of indexes. The $number\_of\_replications$ can be an interval "[0 .. 4]" or an domain identifier defined in **identifiers** block;
- "$< rate >$" defines the rate of the event. It can be a numerical value or an expression identifier declared in the **identifiers** block;
- "$automata[exp - replica]$" is the set of automata where the event appear. This set can be defined using an expression or an interval.

2.3 Reachability Function

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.

Format of the **reachability** block

$\{partial\}$ **reachability** $=<exp>$;

2.4 Network Description

The **network** block is the major component of the SAN description and has an 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.

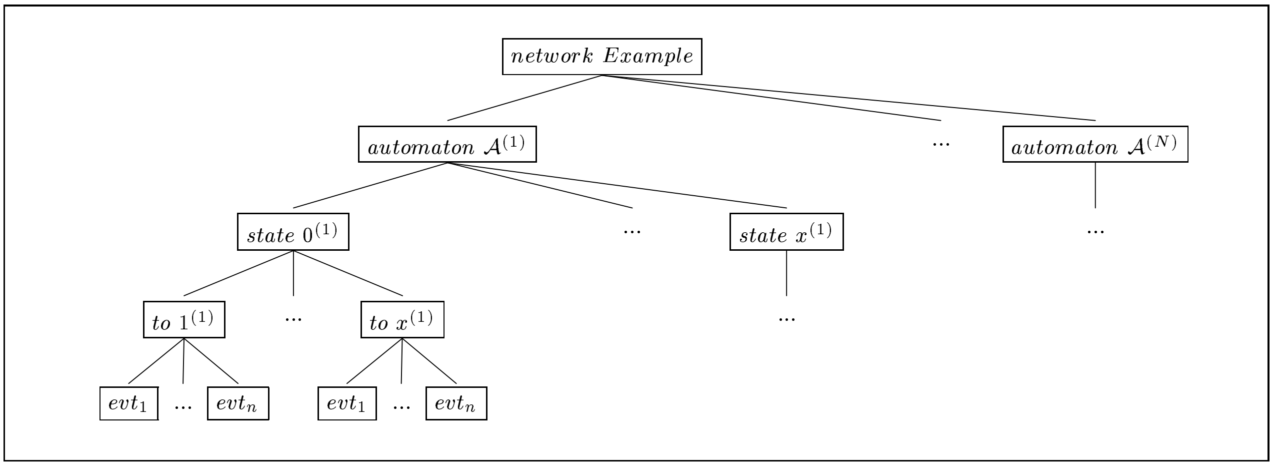

Figure 2.2: Structure of network hierarchy.

The first level, named "**network**", includes general information such as the name of the model and the type of time scale of the model.

Format of the **network** block

---

network $<net\_name>$ $(<type>)$
  aut $<aut\_name>$ $\{[number\_of\_replications]\}$
    stt $<stt\_name>$ $\{[number\_of\_replications]\}$ $\{(reward)\}$
      to($<stt\_name>$)
        $<evt\_name>$ $\{(<prob>)\}$
  ... // *Automata description*

- "$<net\_name>$" is the name of the model. It is a string of alphanumeric characters beginning with a letter;

- "$< type >$" is the time scale of the model. Two type are possible: "*continuous*" or "*discrete*". Currently, only "*continuous*" model analysis is available in PEPS 2003.

The following subsections provide further detail on each of the levels of the network description.

2.4.1 Automaton Description

In this level, each automaton is decribed. The delimiter of the automaton is the reserved word "*aut*" and the name of the automaton. 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 is the name of the replicated automaton, then $A[i]$ is the identifier of one automaton.

Format of the *aut* block

$aut < aut\_name > \{[domain]\}$
  $stt < stt\_name > \{[number\_of\_replications]\}\ \{(reward)\}$
    $to(< stt\_name >)$
      $< evt\_name > \{(< prob >)\}$
  $\ldots$ // *States description*

- "$< aut\_name >$" is the name of the automaton. It is an alphanumeric identifier and it may be used for expression definitions;
- "$[number\_of\_replications]$" is a set of indexes. The $number\_of\_replications$ can be an interval "[0 .. 4]" or a domain identifier. This identifier must defined in **identifiers** block.

2.4.2 State Description

The *stt* section defines a local state of the automaton.

Format of the *stt* block

$stt < stt\_name > \{[number\_of\_replications]\}\ \{(reward)\}$
  $to(< stt\_name >)$
    $< evt\_name > \{(< prob >)\}$
  $\ldots$ // *Transitions description*

- "$< stt\_name >$" is the state identifier, which might be used in function evaluation. PEPS uses an internal index for the states for each automaton. The first declared state of an automaton gets the internal index (also called internal state id) zero, the second gets 1 and so on;
- "$[number\_of\_replications]$" is the number of times that the state appears in the automaton. It is described by an interval [1..4] or by a domain identifier defined in **identifiers** block;
- "$(reward)$" is optional and specifies the state reward. When it is not specified, PEPS gives a default value to the reward, which is the internal state index.

From **Description**

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.

Format of the *from* block

$from < stt\_name >$
  $to(< stt\_name >)$
    $< evt\_name > \{(< prob >)\}$
  $\ldots$ // *Transitions description*

- "$< stt\_name >$" is a state identifier defined in *stt* section. Use "$[x]$" after the state identifier to appoint a specific state in a group of replicated states. "x" can be an identifier previously defined in **identifiers** block or an integer value.

2.4.3 Transition Description

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. "$[x]$" can be used to appoint to a specific replicated state. 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.

Format of the *to* block

$to(< stt\_name >)$
  $< evt\_name > \{(< prob >)\}$
  $\ldots$ // *Events description*

2.4.4 Event Description

Finally, for each transition defined, a set of events (local and synchronizing) that can triggers the transition can be expressed by their names and optionally the probability of occurrence.

Format of the *event* block

$< evt\_name > \{(< prob >)\}$

- "$< evt\_name >$" is the event name that trigger a transition. The event name must have been previously defined in **events** block. Use "[x]" after the $< evt\_name >$ to specify a replicated event;

- "($< prob >$)" is the routing probability for this event. If only one destination is possible, this information can be omitted. The $< prob >$ can be a real value or an expression identifier defined in **identifiers** block.

2.5 Results Description

In this block, the functions used to compute performance indexes of the model are defined. The results given by PEPS are the integral values of these functions with the stationary distribution of the model. This module is **optional**.

Format of the **results** block
results $< res\_name >=< exp >$ **;**

- $< res\_name >$ is a single identifier;
- $< exp >$ is a mathematical expression. The mathematical expressions are described in Section 2.6.

2.6 Mathematical Expressions

This section presents the possibilities that **PEPS** provides to build expressions and functions. 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, but can also be automaton or state identifiers. In this last case, the expressions are functions defined on S the SAN state space. For example, "the number of automata in state n0" (which gives an integer result) is an expression which can be expressed when using PEPS, as well as "the number of automata in state n0 equal 4" (which gives a boolean result).

The format of the operators is summarized in table 2.1. The internal format used in PEPS solver is given in the fourth column, and it is useful only when debugging PEPS. The arithmetic operators are "$+$", "$-$", "$*$", "$/$", "max" and "min" and are not typed (integer or real values). **PEPS** expressions do not have operators priorities and are evaluated from the left to the right. To specify priorities, it is necessary to use brackets. For example, $5 + 6 * 7$ is computed as $(5 + 6) * 7$ in PEPS.

The relational operators are "$==$", "$!=$", "$<$", "$>$", "$<=$", "$>=$". Their result is 1 (coding for TRUE) if the relation is verified and 0 (coding for FALSE) otherwise. The logical operators are: "not" coded with "!", "or" with "||", "and" with "&&", "XOR" with $\wedge$. .

As we already mentioned, the arguments of these operators can be constant values (input of the model), but also functions of the SAN state. With the idea that a SAN has a state which evolves with time, we use the term "current state". These functions are used to express SAN behavior such as "the rate of this event is 0 if the SAN is in state x and equal to r otherwise", or to compute performance index by integration of a function such as "the number of automata in state 0". Before proceeding to these functions, let us describe the way names (identifiers) are handled in PEPS.

- a reference to an automaton identifier is translated in PEPS into a reference to the automaton internal index. This internal index is computed according to the declaration order in the

Network block. The first automaton of the network has internal index zero, the second 1 and so on. Replicated automata are internally numbered using the index of the first one as a base and then incrementing it with the replication index. For example, assume a SAN with **automaton test1** replicated in the interval [1..2], **automaton test2** without replication and **automaton test3** replicated 3 times ([0..2]) , then we have the external identifiers: **test1[1]**, **test1[2]**, **test2**, **test3[0]**, **test3[1]** and **test3[2]**. The internal indexes are: 0 for **test1[1]**, 1 for **test1[2]**, 2 for **test2**, 3 for **test3[0]**, 4 for **test3[1]** and 5 for **test3[2]**. This internal numbering allows to defined subsets of automata as intervals. The interval (**test1[2]**, **test3[1]**) is exactly the subset of automata: **test1[2]**, **test2**, **test3[0]**, **test3[1]**.

- a reference to a state identifier is also replaced in PEPS with a reference to an automaton index. The external identifiers correspond to an internal index computed according to the declaration order and the number of replications. For example, if the automaton **test1** has 3 states, named **A**, **B** and **C** declared in this order, the internal index of **A** is 0, **B** is 1 and **C** is 2. If there are replications, the corresponding offset is applied. For a single queue, a state is replicated, and it works as follows: take a queue with a block of replicated states. This block has one state **rep** replicated four times. Then, the internal index ranges from 0 to 3 in the order : **rep[0]**, **rep[1]**, **rep[2]**, and **rep[3]**, but the first **rep[0]** and the last **rep[3]** have some transitions to outside the group ignored.
- PEPS maintains a global (for all the automata) table giving a correspondence between a state external identifier and state internal index (called name-index table). If several distinct automata have the same state identifier, a warning is output if and only if these states do not have the same internal index. This situation might lead to errors if these state identifiers do not correspond to the same internal index (internal indexes are computed per automaton). So the user should be aware of this implementation when writing a function based on external identifiers as "number of automata in state n0".

These functions are:

- **st** $<$ *aut_name* $>$: gives the current state of automaton $<$ *aut_name* $>$. $<$ *aut_name* $>$ is an external identifier, and the output of this function is an internal state index. Another (historical) syntax for this function is **@** $<$ *aut_name* $>$.
- **nb** $<$ *stt_name* $>$: gives the total number of automata of the SAN in the state $<$ *stt_name* $>$. $<$ *stt_name* $>$ is an external identifier and PEPS translates it into an internal index. The user should be careful when using this function: it might lead to errors when there are automata having a state named $<$ *stt_name* $>$ but with different internal index, or when there are automata without a state named $<$ *stt_name* $>$ (PEPS gives a warning in this case).
- **nb** [$<$ *aut_name* $>$ .. $<$ *aut_name* $>$] $<$ *stt_name* $>$: it is an extension of the preceding function, but the count is done on the automata interval [$<$ *aut_name* $>$ .. $<$ *aut_name* $>$]. The notion of "interval" refers to the total ordering described above. This notation is very useful when an automaton is replicated and the interval is exactly all the replicated automata.
- **rw** $<$ *aut_name* $>$: in SAN, rewards may be associated to states. This function gives the current reward of automaton $<$ *aut_name* $>$. $<$ *aut_name* $>$ is an external identifier, and the output of this function is a reward, thus a real or integer value. This function is very similar to **st** $<$ *aut_name* $>$. Remember that if the reward is not explicitly given by the user, the internal state index is used as a reward. This makes sense when the states coded 0, 1, 2, etc. are the number of customers in a queue. In this case the two functions **rw** $<$ *aut_name* $>$ and **st** $<$ *aut_name* $>$ are identical.

- **sum_rw** $[< aut\_name > .. < aut\_name >]$: gives the sum of the current rewards of the automata in the interval $[aut\_name > .. < aut\_name >]$.

- **sum_rw** $[< aut\_name > .. < aut\_name >] < stt\_name >$: gives the sum of the rewards of the automata in the interval $[< aut\_name > .. < aut\_name >)$ which are in the state $< stt\_name >$.

- **prod_rw** $[< aut\_name > .. < aut\_name >]$ and
  prod_rw $[< aut\_name > .. < aut\_name >] < stt\_name >$: are similar to the preceding functions, with a change of operator.

External Format	Semantic of the Format	Example
State and Automaton Operators		
st $< aut\_name >$	gives the current state of automaton "$< aut\_name >$"	st processA
@ $< aut\_name >$	the same as above, another notation	@processA
nb $< stt\_name >$	gives the total number of automata in the state "$< stt\_name >$"	nb util
nb $[< aut\_name > .. < aut\_name >] < stt\_name >$	for the automata in the interval "$[< aut\_name > .. < aut\_name >]$", gives the total number of automata in the state "$< stt\_name >$"	nb [processA .. processD) util
rw $< aut\_name >$	gives the reward associated with the current state of automaton "$< aut\_name >$"	rw processA
sum_rw $[< aut\_name > .. < aut\_name >]$	gives the sum of the rewards of the current states of the automata in the interval $[< aut\_name > .. < aut\_name >]$	sum_rw [processA .. processD]
sum_rw $[< aut\_name > .. < aut\_name >] < stt\_name >$	gives the sum of the rewards of the automata in the interval "$< aut\_name > .. < aut\_name >$", which are in the state $< stt\_name >$	sum_rw [processA .. processD] util
prod_rw $[< aut\_name > .. < aut\_name >]$	product of the rewards of the current of the automata of interval "$< aut\_name > .. < aut\_name >$"	prod_rw [processA .. processD]
prod_rw $[< aut\_name > .. < aut\_name >]< stt\_name >$	product of the rewards of the current states of the automata of interval "$< aut\_name > .. < aut\_name >$", which are in the state $< stt\_name >$	product_rw [processA .. processD] util
Arithmetic Operators		
$< exp1 > + < exp2 >$	sum of "$< exp1 >$" and "$< exp2 >$"	$5\ +\ 3$
$< exp1 > - < exp2 >$	substraction of "$< exp2 >$"minus "$< exp1 >$"	$5\ -\ 3$
$< exp1 > * < exp2 >$	product of "$< exp1 >$" and "$< exp2 >$"	$5\ *\ 3$
$< exp1 > / < exp2 >$	division of "$< exp1 >$" by "$< exp2 >$"	$5\ /\ 3$
min $[< exp1 > .. < exp2 >]$	gives the miniimum of two arguments	min[5 .. 3]
max $[< exp1 > .. < exp2 >]$	gives the maximum of two arguments	max[5 .. 3]
Relational Operators		
$< exp1 > == < exp2 >$	if "$< exp1 >$" is equal to "$< exp2 >$", gives true ("1") else false ("0")	$5\ ==\ 3$
$< exp1 > != < exp2 >$	if "$< exp1 >$" is not equal to "$< exp2 >$", givess true ("1") else false ("0")	$5\ !=\ 3$
$< exp1 > < < exp2 >$	if "$< exp1 >$" is smaller than "$< exp2 >$", gives true ("1") else false ("0")	$5\ <\ 3$
$< exp1 > <= < exp2 >$	if "$< exp1 >$" is smaller or equal than "$< exp2 >$", gives true ("1") else false ("0")	$5\ <=\ 3$
$< exp1 > > < exp2 >$	if "$< exp1 >$" is greather than "$< exp2 >$", results true ("1") else results false ("0")	$5\ >\ 3$
$< exp1 > >= < exp2 >$	if "$< exp1 >$" is greater or equal than "$< exp2 >$", gives true ("1") else false ("0")	$5\ >=\ 3$
Logical Operators		
$! < exp >$	logical "not"	$5\ >\ 3)\ !\ (7\ <\ 8)$
$< exp1 > \vert\vert < exp2 >$	logical "or"	$(5\ >\ 3)\ \vert\vert\ (7\ <\ 8)$
$< exp1 > \&\& < exp2 >$	logical "and"	$(5\ >\ 3)\ \&\&\ (7\ <\ 8)$
$< exp1 > \wedge < exp2 >$	logical "XOR"	$(5\ >\ 3)\ \wedge\ (7\ <\ 8)$

Table 2.1: Operators

Chapter 3

Modeling Examples

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.

3.1 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.

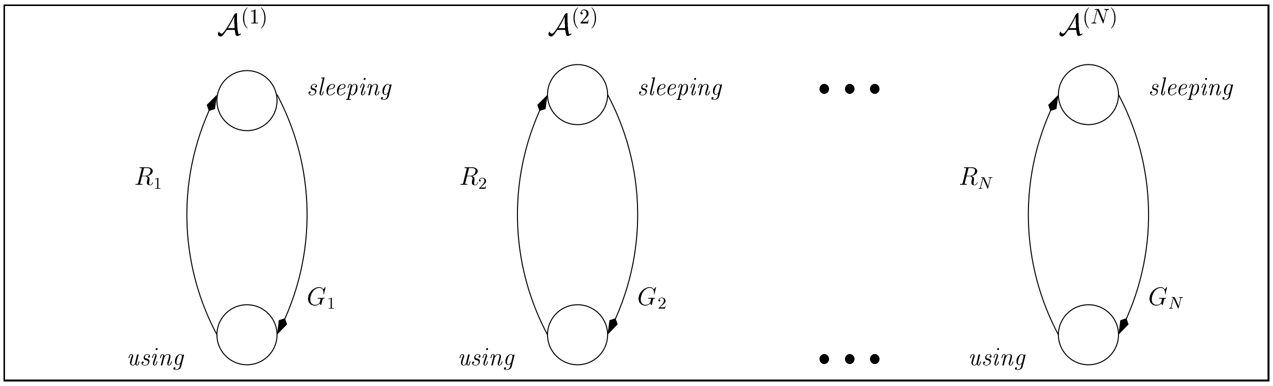

Figure 3.1: Resource Sharing Model - version 1

In our SAN representation (Figure 3.1), 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. Figure 3.1 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[N] // local event G has rate f and appears in all automata P
   loc R  (mu) P[N] // 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[N]
    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 manual, 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). Figure 3.2 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
```

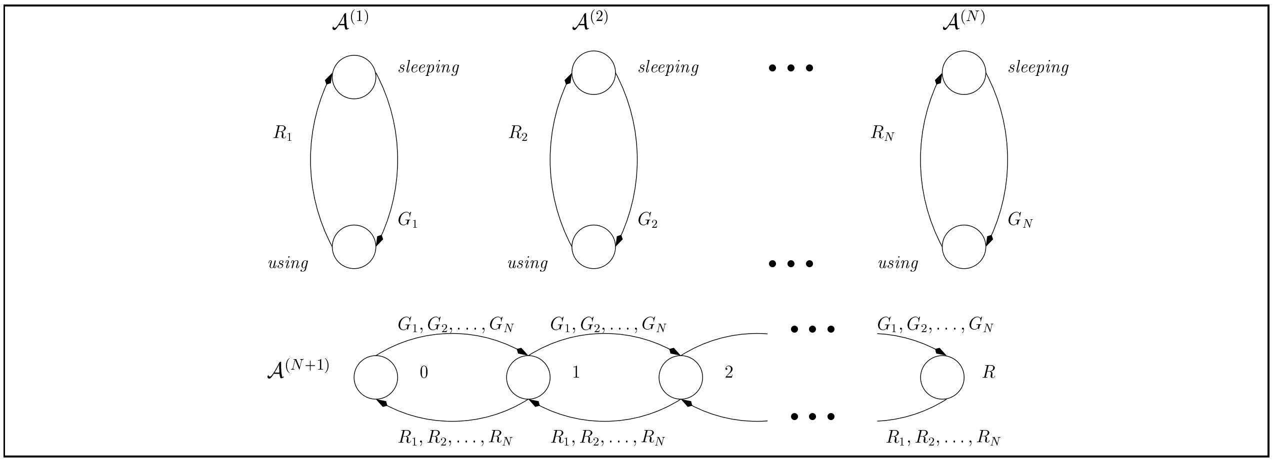

Figure 3.2: Resource Sharing Model without functions - version 2

```
   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

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 resources must be equal to number
   //  of occupied resources in the Resource 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
```

3.2 First Server Available Queue

The second example considers a queue with common exponential arrival and a finite number (C) of distinguishable and ordered servers ($C_i, i = 1 \dots C$). 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 SAN model describing this queue is presented in Figure 3.3. 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.

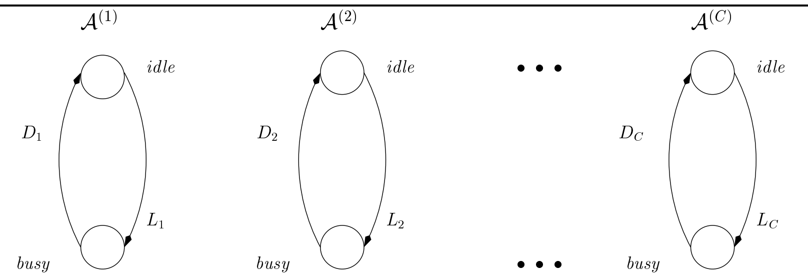

Figure 3.3: First Server Available Model

The PEPS 2003 textual formats for this model is 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;
```

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 this alternative model is as follows:

```
//============================================== FSA 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;
```

3.3 Example: A Mixed Queueing Network

The final example is a mixed queueing network (Fig. 3.4) 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.

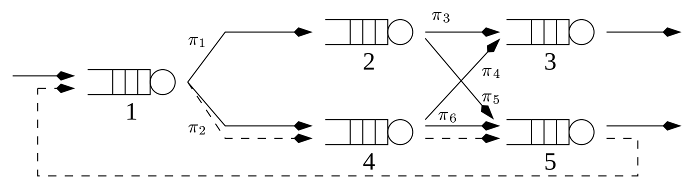

Figure 3.4: Mixed queueing network

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 equivalent SAN model has a product state space containing 28,579,716 states of which only 402,732 are reachable. This model is described as follow. More detailes can be obtained from the PEPS web page [16].

```
//============================================== MQN model ========================================
//                                          Mixed Queueing Network
//=================================================================================================
identifiers
  N2 = 10;

  K1 = [0..10];
  K2 = [0..5];
  K3 = [0..5];
  K4 = [0..8];
  K5 = [0..8];

  k1 = 10;
  k2 = 5;
  k3 = 5;
  k4 = 8;
  k5 = 8;

  Sri00 =   0.1;
  Sri01 =   0.2;
  Sri02 =   0.3;
  Sri03 =   0.4;
  Sri04 =   0.5;
  Sri10 =   0.1;
  Sri13 =   0.4;
  Sri14 =   0.5;

  Lri00 =   1;

  Mu00 =   1/Sri00;
  Mu01 =   1/Sri01;
  Mu02 =   1/Sri02;
  Mu03 =   1/Sri03;
  Mu04 =   1/Sri04;
  Mu10 =   1/Sri10;
  Mu13 =   1/Sri13;
  Mu14 =   1/Sri14;

  F_queue1 =   ( st class1_queue1 + st class2_queue1 ) < k1;
  F_queue2 =   ( st class1_queue2 ) < k2;
  F_queue3 =   ( st class1_queue3 ) < k3;
  F_queue4 =   ( st class1_queue4 + st class2_queue4 ) < k4;
  F_queue5 =   ( st class1_queue5 + st class2_queue5 ) < k5;
  FL_class1_queue1 =   Lri00 * F_queue1;

  n1_c1 = st class1_queue1/(st class1_queue1 + st class2_queue1);
  n1_c2 = st class2_queue1/(st class1_queue1 + st class2_queue1);
  n4_c1 = st class1_queue4/(st class1_queue4 + st class2_queue4);
  n4_c2 = st class2_queue4/(st class1_queue4 + st class2_queue4);
  n5_c1 = st class1_queue5/(st class1_queue5 + st class2_queue5);
  n5_c2 = st class2_queue5/(st class1_queue5 + st class2_queue5);

  rot_class1_queue1toqueue2 =   0.5 * Mu00 * n1_c1;
  rot_class1_queue1toqueue4 =   0.5 * Mu00 * n1_c1;
  rot_class1_queue2toqueue3 =   0.5 * Mu01;
  rot_class1_queue2toqueue5 =   0.5 * Mu01;
  rot_class1_queue4toqueue3 =   0.5 * Mu03 * n4_c1;
  rot_class1_queue4toqueue5 =   0.5 * Mu03 * n4_c1;
  rot_class2_queue1toqueue4 =   1 * Mu10 * n1_c2;
  rot_class2_queue4toqueue5 =   1 * Mu13 * n4_c2;
  rot_class2_queue5toqueue1 =   1 * Mu14 * n5_c2;
  rot_class1_queue3toOUT =    Mu02;
  rot_class1_queue5toOUT =    Mu04 * n5_c1;

events
  loc l_FL_class1_queue1 (FL_class1_queue1)
  loc l_rot_class1_queue3toOUT (rot_class1_queue3toOUT)
  loc l_rot_class1_queue5toOUT (rot_class1_queue5toOUT)
  syn s_class1_queue1toqueue2 (rot_class1_queue1toqueue2)
  syn s_class1_queue1toqueue4 (rot_class1_queue1toqueue4)
  syn s_class1_queue2toqueue3 (rot_class1_queue2toqueue3)
```

```
  syn s_class1_queue2toqueue5 (rot_class1_queue2toqueue5)
  syn s_class1_queue4toqueue3 (rot_class1_queue4toqueue3)
  syn s_class1_queue4toqueue5 (rot_class1_queue4toqueue5)
  syn s_class2_queue1toqueue4 (rot_class2_queue1toqueue4)
  syn s_class2_queue4toqueue5 (rot_class2_queue4toqueue5)
  syn s_class2_queue5toqueue1 (rot_class2_queue5toqueue1)

reachability = ((st class1_queue1 + st class2_queue1)<= k1)
            && ((st class1_queue4 + st class2_queue4)<= k4)
            && ((st class1_queue5 + st class2_queue5)<= k5)
            && ((st class2_queue1 + st class2_queue4 + st class2_queue5) == N2);

network mqn (continuous)
  aut class1_queue1
    stt n[K1]  to(++)  l_FL_class1_queue1
               to(--)  s_class1_queue1toqueue2
                       s_class1_queue1toqueue4

  aut class1_queue2
    stt n[K2] to(++)  s_class1_queue1toqueue2
              to(--)  s_class1_queue2toqueue3
                      s_class1_queue2toqueue5

  aut class1_queue3
    stt n[K3]  to(++)  s_class1_queue2toqueue3
                       s_class1_queue4toqueue3
               to(--)  l_rot_class1_queue3toOUT

  aut class1_queue4
    stt n[K4] to (++)  s_class1_queue1toqueue4
              to (--)  s_class1_queue4toqueue3
                       s_class1_queue4toqueue5

  aut class1_queue5
    stt n[K5] to (++)  s_class1_queue2toqueue5
                       s_class1_queue4toqueue5
              to (--)  l_rot_class1_queue5toOUT

  aut class2_queue1
    stt n[K1] to (++)  s_class2_queue5toqueue1
              to (--)  s_class2_queue1toqueue4

  aut class2_queue4
    stt n[K4] to (++)  s_class2_queue1toqueue4
              to (--)  s_class2_queue4toqueue5

  aut class2_queue5
    stt n[K5] to (++)  s_class2_queue4toqueue5
              to (--)  s_class2_queue5toqueue1

results
  nri00 =   st class1_queue1;
  nri01 =   st class1_queue2;
  nri02 =   st class1_queue3;
  nri03 =   st class1_queue4;
  nri04 =   st class1_queue5;
  nri10 =   st class2_queue1;
  nri13 =   st class2_queue4;
  nri14 =   st class2_queue5;
```

Chapter 4

Using PEPS 2003

In this chapter we present a brief description of the installation procedure for the PEPS software (Section 4.1). Next, the basic functions of the software interface is presented in Section 4.2.

4.1 PEPS 2003 Installation

Being a academic open-source software, the PEPS 2003 installation is quite simple and do not demand any special feature from your system. The *Makefile* do not perform any test to verify the availability of the GNU C++ compiler in the system. This compiler is needed to compile the PEPS software itself during the installation, but also during the execution. One of the new numerical improvements of the PEPS 2003 version is the compilation of functional elements *just-in-time* [3]. This procedure calls the `g++` compiler, so make sure this command is reachable from your environment (type `which g++` at your prompt to verify it).

4.1.1 Getting the source

Retrieve from our site the file `Peps2003.tgz`, use `tar -xvzf Peps2003.tgz` to uncompress this file and to generate the directory `Peps2003` and all the hierachy of files.

4.1.2 Compiling

Enter the PEPS 2003 directory and edit the *Makefile* (uncomment the 28th line and comment the 29th line) if you want to generate a debug version of PEPS 2003 (see Section 4.2.9). Usually, normal users do not need to change that option, which is by the default the generation of an optimized version of PEPS 2003 (28th line commented and 29th line uncommented). Once you did it, just type `make` and the binary files of PEPS will be generated.

4.1.3 Binary files

In the directory `./Peps2003/bin/` there is a file named `peps` which is the executable. To run Peps just type `./Peps2003/bin/peps`. If you want to run Peps from anywhere, add the directory `./Peps2003/bin/` in your standard `PATH` or put a link using Unix command `ln`. The first time Peps is used in a given directory, it will initialize all auxiliary subdirectories.

4.2 PEPS 2003 User Interface

When the PEPS sofware package is initiated, by typing `peps`, the following message is displayed:

```
   +----------------------------------------------------+
   |             This is PEPS 2003 - the SAN tool        |
   |  released on: May 21 2003 -- compiled on: Sep  1 2003  |
   +----------------------------------------------------+

  1) Compile a SAN model            5) Load/Show/Remove data structures
  2) Solve a compiled SAN model     6) Inspect data structures
  3) Probability vector facilities  7) Sparse matrix facilities
  4) Preferences                    8) About this version

   0) Exit PEPS (Option 0 always exits the current menu)
```

Options 1), 2) and 4) are used to solve a model, options 3), 5) and 6) are used to manage data, and 7) is the bridge to sparse technology (MARCA). In any menu, typing 0 results in quitting the current level. In the following, we proceed through the various items of this first menu. When using PEPS, it is probably best to have a directory per model. In general PEPS stores files in the current directory. Thus, if PEPS is called from a model directory, all related files are stored in this directory.

4.2.1 Compiling a SAN model

Choosing the option `Compiling a SAN Model` leads to the following menu.

```
             ******* Compiling a SAN Model *******

 1) Standard Compilation
 2) Compilation with Algebraic Aggregation
 3) Compilation with Aggregation of Replicas
```

The abbreviations `san, des, dsc, agg` and `cnd` correspond to various formats for a SAN model and are stored in files with the corresponding suffix in subdirectories of the current directory. The normal use of PEPS is to start by editing a `toto.san` file which contains a textual definitions of the SAN named `toto`. Option 6) creates, from the file `toto.san` , the files `toto.des` , `toto.dsc` , and `toto.cnd` corresponding respectively to the textual version of the descriptor, an internal version of the descriptor, and a normalized version of the descriptor (normalizing uses the technique of uniformization to transform the generator into a probability transition suitable for solution by an iterative method). Option 6) prints the following output onto screen (assuming that the file `toto.san` does not contain errors):

```
Compilation of a SAN model
Enter textual SAN file name: mutex
Compile_Network
 :-) file 'des/toto.des' saved
```

```
 :-) file 'des/toto.dic' saved
 :-) file 'des/toto.tft' saved
 :-) file 'des/toto.fct' saved
 :-) file 'des/toto.res' saved
Compile_Function_Table
 :-) file 'des/toto.tft' read
 :-) file 'dsc/toto.ftb' saved
Compile_Descriptor
 :-) file 'des/toto.des' read
 :-) file 'dsc/toto.dsc' saved
Compile_Reacheable_SS
 :-) file 'dsc/toto.rss' saved
 :-) file 'des/toto.fct' read
Compile_Dictionary
 :-) file 'dsc/toto.dct' saved
 :-) file 'des/toto.dic' read
Compile_Integration_Function
 :-) file 'des/toto.res' read
 :-) file 'dsc/toto.inf' saved
 :-) file 'cnd/toto.cnd' saved
 :-) file 'cnd/toto.ftb' saved
 :-) file 'cnd/toto.rss' saved
 :-) file 'jit/peps_jit.C' saved

Translation performed:  compilation of a SAN model
  (largest element in reachable states: 1.2000000000000000e+01)
 - user time spent:     -4.1674021004911577e-19 seconds
 - system time spent:   -4.1674021004911577e-19 seconds
 - real time spent:     1.3387849546852522e+00 seconds
```

The user inputs a file name (which may be a path like `example/toto`) but without the suffix `san`. This rule holds for all input files in PEPS: they are to be input without suffix. The files `fct`, `tft`, `dic`, and `res` correspond respectively to the reachability function, the table for functions appearing in the `identifiers` section, the dictionary for all names in the SAN model, the table for the functions appearing in the `results` section, all in textual format. The files `rss`, `ftb`, `dct`, and `inf` correspond respectively to the reachability state space (coded as a compact vector with 0 or 1 as entries), the table for functions appearing in the `identifiers` section, the dictionary for all names in the SAN model, the table for the functions appearing in the `results` section, all in internal format. Final comments give some performance information on the tool and the normalizing factor (`largest element in reachable states`).

Aggregation is an extra step, and usually improves performance significantly. Option 2) is similar to option 1) but with an aggregation step performed after the generation of `dsc`. It requires an input from the user as shown in the following scenario.

```
Compilation of a SAN model
Enter textual SAN file name: toto
Compile_Network
 :-) file 'des/toto.des' saved
 :-) file 'des/toto.dic' saved
```

```
 :-) file 'des/toto.tft' saved
 :-) file 'des/toto.fct' saved
 :-) file 'des/toto.res' saved
Compile_Function_Table
 :-) file 'des/toto.tft' read
 :-) file 'dsc/toto.ftb' saved
Compile_Descriptor
 :-) file 'des/toto.des' read
 :-) file 'dsc/toto.dsc' saved
Compile_Reacheable_SS
 :-) file 'dsc/toto.rss' saved
 :-) file 'des/toto.fct' read
Compile_Dictionary
 :-) file 'dsc/toto.dct' saved
 :-) file 'des/toto.dic' read
Compile_Integration_Function
 :-) file 'des/toto.res' read
 :-) file 'dsc/toto.inf' saved
Enter contiguous integer values to identify the groups starting from '0'
 (last group number must be less than '8')
Automaton process[1] will be in group (0..7): 0
Automaton process[2] will be in group (0..7): 0
Automaton process[3] will be in group (0..7): 1
Automaton process[4] will be in group (0..7): 1
Automaton process[5] will be in group (0..7): 2
Automaton process[6] will be in group (0..7): 2
Automaton process[7] will be in group (0..7): 2
Automaton process[8] will be in group (0..7): 2

You need a new file name to the aggregate model
Enter descriptor file name: group3
 :-) file 'agg/group3.dsc' saved
 :-) file 'agg/group3.rss' saved
 :-) file 'agg/group3.ftb' saved
 :-) file 'cnd/group3.cnd' saved
 :-) file 'cnd/group3.ftb' saved
 :-) file 'cnd/group3.rss' saved
 :-) file 'jit/peps_jit.C' saved

Translation performed:  compilation of a SAN model
  (largest element in reachable states: 1.2000000000000000e+01)
 - user time spent:     1.1096131609031334e-18 seconds
 - system time spent:   -2.0837010502455788e-19 seconds
 - real time spent:     2.8550801525125280e-01 seconds
```

In the dialog for specifying the groups, PEPS output the name of all automata. In this example they are named `process[1]`, `process[2]`, etc. and the user input a group index. The groups are identified by contiguous numbers starting from 0.

The option 3) is quite similar to option 2) but the aggregation step automatic. The PEPS 2003 performed the aggregation of replicas. It do not requires an input from the user as shown in the following scenario.

```
Compilation of a SAN model
Enter textual SAN file name: toto
Compile_Network
 :-) file 'des/toto.des' saved
 :-) file 'des/toto.dic' saved
 :-) file 'des/toto.tft' saved
 :-) file 'des/toto.fct' saved
 :-) file 'des/toto.res' saved
Compile_Function_Table
 :-) file 'des/toto.tft' read
 :-) file 'dsc/toto.ftb' saved
Compile_Descriptor
 :-) file 'des/toto.des' read
 :-) file 'dsc/toto.dsc' saved
Compile_Reacheable_SS
 :-) file 'dsc/toto.rss' saved
 :-) file 'des/toto.fct' read
Compile_Dictionary
 :-) file 'dsc/toto.dct' saved
 :-) file 'des/toto.dic' read
Compile_Integration_Function
 :-) file 'des/toto.res' read
 :-) file 'dsc/toto.inf' saved

You need a new file name to the aggregate model
Enter descriptor file name: group1
 :-) file 'agg/group1.dsc' saved
 :-) file 'agg/group1.rss' saved
 :-) file 'agg/group1.ftb' saved
 :-) file 'cnd/group1.cnd' saved
 :-) file 'cnd/group1.ftb' saved
 :-) file 'cnd/group1.rss' saved
 :-) file 'jit/peps_jit.C' saved

Translation performed:  compilation of a SAN model
  (largest element in reachable states: 1.2000000000000000e+01)
 - user time spent:     -2.0837010502455788e-19 seconds
 - system time spent:   -2.0837010502455788e-19 seconds
 - real time spent:     2.0523505419259891e-01 seconds
```

4.2.2 Solving a SAN model

Choosing option 2), `Solve a compiled SAN model` from the main menu leads to:

```
       ******* Solving a SAN model *******

No Preconditioning:      Diagonal Preconditioning:

 1) Power Method             4) Power Method
 2) Arnoldi Method           5) Arnoldi Method
 3) GMRES Method             6) GMRES Method
```

Any solution method is applied to the model currently in memory: it is the last model which has just been compiled and normalized, or the last model loaded (with option 5) in the main menu). If none of this has been done, the user is asked to load a `.cnd` file (without suffix or path).

Two types of iterative methods are possible: with or without diagonal preconditionning. If the user chooses option 1) after compiling (with algebraic aggregation) the model `toto.san` then the following dialog occurs:

```
Solution of the model 'cnd/group3.cnd' (3 automata - 37/176 states)

Enter vector file name: group3
Iteration 0: largest: 1.3333333333333336e-01 (0) smallest: 4.5045045045044932e-04 (3)
Iteration 10: largest: 8.7582923953277170e-01 (0) smallest: 2.4537491691937498e-04 (3)
Iteration 20: largest: 8.7633846991621989e-01 (0) smallest: 2.4342891499872719e-04 (3)
Iteration 30: largest: 8.7633885073739015e-01 (0) smallest: 2.4342745970646755e-04 (3)
Iteration 32:
Power solution
Number of iterations:   33
 - user time spent:     2.0000000000000000e-02 seconds
 - system time spent:   -2.0837010502455788e-19 seconds
 - real time spent:     1.8977033905684948e-02 seconds
Residual Error: 3.4663497766596052e-11 - The method converged (solution found)!
 :-) file 'group3.vct' saved
 :-) file 'group3.tim' saved

Integration (dsc/toto.inf) of the vector 'group3.vct' (176 states)
solution of 'cnd/group3.cnd' (256 states)
Integration of function used : 1.3047711789366598e-01
Integration of function n_used : 7.8695228821063381e+00
```

The first line provides information about the model: the number of automata after grouping (if any), the number of reachable state in the model and the size of the product state space (`37/176 states`). The prompt `Enter vector file name` requests a file name in which to store the stationnary probability vector. Here, the name `group3` is given and the file is called `group3.vct`. Then, the iteration procedure starts, and its progress can be followed on the screen: it prints out the number of iterations (i.e. number of vector-descriptor multiplications), the value of the smallest and largest element (and the corresponding indexes of these elements in the vector). For the power method, the evolution of these two values should be monotonic. When the iterative procedure stops, PEPS prints out performance figures, the residual error and a message indicating whether convergence has been reached. File `group3.tim` contains a performance report for this run. For multiple run, the reports are concatenated. In the `result` section of the model `toto.san` there were two functions to integrate `used` and `n_used`. These results are also output.

4.2.3 Changing preferences

These preference are related to the solution method. When option 4) is chosen from in the main menu, the following possibilities are presented:

```
        ******* Changing Preferences *******

 1) Verbose Mode:                                  on
 2) Maximum number of iterations:                  1000
 3) Stop iteration criterion:                      convergence test
 4) Tolerance accepted:                            1.0000000000000000e-10
 5) Error Test:                                    maximum of individual absolute errors
 6) Initial vector:                                equiprobable
 7) Krylov subspace size (for projection methods): 10
 8) Vector-Descriptor product method :             mixed method
 9) Vector implementation :                        extended (E)

Enter a number to change the corresponding parameter:
```

Each of these options leads to a multiple choice menu. For example, choosing 3) :

```
Enter a number to change the corresponding parameter: 3
Enter a choice: 1 fixed number of iterations
                2 stability test
                3 convergence test
0

Adopting option 3 (convergence test)
```

If 0 is chosen, no change is made. When is first PEPS is initiated, default values are given for each as shown on the menu above.

If option 5) is chosen, the user is presented the following menu:

```
Enter a number to change the corresponding parameter: 5
Enter a choice: 1 maximum absolute individual error
                2 accumulated absolute error
                3 maximum relative individual error
2

Adopting option 2 (accumulated absolute error)
```

If option 6) is chosen, the following choices are possible for the initial iteration vector:

```
Enter a number to change the corresponding parameter: 6
Enter a choice of initial vector: 1 equiprobable
                                  2 approximated
                                  3 user vector
```

Thus these choices allow for 1) a vector on the reachable state space with identical components, 2) a vector with more weight on reachable states corresponding to small diagonal entries (the algorithm takes the inverse of the diagonal entries of the descriptor on the reachable state space and normalizes this vector), 3) a vector stored in a `.vct` file previously defined by the user.

If option 9) is given, the following choices are possible for the vetor implementation:

```
Enter a number to change the corresponding parameter: 9
Enter a choice of vector implementation : 1 extended (E)
                                          2 partially reduced (PR)
                                          3 fully reduced (FR)
```

4.2.4 Probability vector facilities

When option 3) `Probability vector facilities` is chosen from the main menu, the following menu is presented:

```
         ******* Probability Vector Facilities *******

 1) local state probabilities   5) global state probabilities
 2) integrate results           6) compute residue
 3) compare two vectors         7) vector characteristics
 4) inspect probability vector  8) inspect reachable state space

                 9) import a MARCA vector
```

We shall now examine these options. The commands are applied to the last computed or last vector loaded into memory. If such a vector does not exist, the user is asked for an input file (`.vct`). Some commands are applied relative to a model. This model is the last compiled or loaded model. The user should be carefull to have a correct link between the two parts. Nevertheless, the vector may have been computed by MARCA, and the model by PEPS. PEPS checks that both structures corresponde to the same product state space.

Local state probabilities

This option computes the marginal probabilities for an automaton. When choosing 1) PEPS answers:

```
Integration of the vector 'group3.vct' (176 states)
solution of 'cnd/group3.cnd' (256 states)
The automata are:
    0) process[1]
    1) process[2]
    2) process[3]
    3) process[4]
    4) process[5]
    5) process[6]
```

```
    6) process[7]
    7) process[8]

Choose an automaton (enter its index): 3
The local states of automaton process[4] are:
    0) sleeping
    1) using
Choose an option (enter its index or '2' for all states): 2
Assuming all states

   9.8369036026329248e-01 for local state 'sleeping'
   1.6309639736708244e-02 for local state 'using'
 ---------------------------
   1.0000000000000007e+00 (total for automaton 'process[4]')
```

PEPS remember the name of the probability vector file and the automata name of the corresponding model, which is in memory. Then, the user chooses an automaton and then a state for this automaton. If an out of range number is given, PEPS calculates the entire marginal distribution.

Integrate results

For the vector in memory and the functions in the `results` section of the model in memory, the integration is performed (note that when solving a model, integration is performed automatically).

```
Integration (dsc/toto.inf) of the vector 'group3.vct' (176 states)
solution of 'cnd/group3.cnd' (256 states)
Integration of function used : 1.3047711789366598e-01
Integration of function n_used : 7.8695228821063381e+00
```

Compare two vectors

This options compares the vector in memory with a vector given in an input file. An example is given below:

```
There is no Probability Vector in memory
Do you want to: 1) load a vector from a file
                2) create an equiprobable vector
1

Enter vector file name: x
 :-) file 'x.vct' read

Compare vector 'x.vct' (176 states) with
Enter vector file name: z
 :-) file 'z.vct' read
The maximum absolute error is: 0.0000000000000000e+00
The maximum relative error is: 0.0000000000000000e+00
```

```
Vector 'x.vct' reachable states probability
   max.:  1.1280188607687570e-02   min.: 1.1280184151260250e-06
Vector 'z.vct' reachable states probability
   max.:  1.1280188607687570e-02   min.: 1.1280184151260250e-06
```

Inspecting probability vector

This options allows the user to inspect parts of the vector in memory, using index ranges.

```
Vector 'group3.vct' (37/176 states)

Enter first state: 0
Enter last state: 10
0 0 0  : 8.7633885095484709e-01 (0)
0 0 1  : 1.4605647524579988e-02 (1)
0 0 2  : 1.4605647524579988e-02 (2)
0 0 3  : 2.4342745887546483e-04 (3)
0 0 4  : 1.4605647524579988e-02 (4)
0 0 5  : 2.4342745887546483e-04 (5)
0 0 6  : 2.4342745887546483e-04 (6)
0 0 7  : 1.4605647524579988e-02 (7)
0 0 8  : 2.4342745887546483e-04 (8)
0 0 9  : 2.4342745887546483e-04 (9)
0 0 10  : 2.4342745887546483e-04 (10)
Do you need state name vs. number table (y/n)? n

Do you wish to see more elements (y/n)? n
```

The output gives the values of the probability vector and the coding of the global state as the vector of the local states.

Global state probabilities

This option allows the user to select an entry in a vector using its coding in terms of a vector of local states:

```
Inspecting the probability of one global state of the vector group3.vct

Do you wish to: 1) inspect a probability indicating its index
                2) inspect a probability indicating its local states vector
     (as always in PEPS, option '0' leaves the current menu)
2
Enter the local state of each of the automata
The local states of automaton process[1] are:
```

```
    0) sleeping
    1) using
Choose a state (enter its index): 0
The local states of automaton process[2] are:
    0) sleeping
    1) using
Choose a state (enter its index): 0
The local states of automaton process[3] are:
    0) sleeping
    1) using
Choose a state (enter its index): 0
The local states of automaton process[4] are:
    0) sleeping
    1) using
Choose a state (enter its index): 0
The local states of automaton process[5] are:
    0) sleeping
    1) using
Choose a state (enter its index): 1
The local states of automaton process[6] are:
    0) sleeping
    1) using
Choose a state (enter its index): 1
The local states of automaton process[7] are:
    0) sleeping
    1) using
Choose a state (enter its index): 1
The local states of automaton process[8] are:
    0) sleeping
    1) using
Choose a state (enter its index): 1
-------------------------------------------------------------------------
Global State: 0: 8.7633885095484709e-01 (after grouping: 0)
[ 0 0 0 ] is reachable
-------------------------------------------------------------------------
Do you wish to: 1) inspect a probability indicating its index
                2) inspect a probability indicating its local states vector
     (as always in PEPS, option '0' leaves the current menu)
0
```

Compute residue

Given a vector and a model, this computation allows the user to verify that the vector is a solution of the model.

```
Computing the residual of the solution 'x.vct' (37/176 states) for the
descriptor cnd/gropu3.cnd (3 automata)
Please enter a probability vector name in which to store the residual
```

```
Enter vector file name: res
 :-) file 'res.vct' saved
The largest element in residual vector is 3.4663497766596052e-11 (0),
therefore the vector x.vct is the approximated solution
to the descritor cnd/gropu3.cnd (smallest: 3.8693071641691945e-12 (11)
```

Vector characteristics

This option provides some information about a vector. It facilitates a quick overview of a vector.

```
Do you wish to: 1) load a vector from a file
                2) create an equiprobable vector
                3) use the current vector x.vct
3

Vector 'x.vct' (37/176 states)
Sum of all 176 elements: 1.0000000000000009e+00
  - Largest  reachable element (0): 8.7633885095484709e-01
  - Smallest reachable element (3): 2.4342745887546483e-04
First elements: 8.7633885095484709e-01 1.4605647524579988e-02
                1.4605647524579988e-02 2.4342745887546483e-04
Last elements: 0.0000000000000000e+00 0.0000000000000000e+00
               0.0000000000000000e+00 0.0000000000000000e+00
```

Inspect reachable state space

This option is similar to 4) but applies only on the reachable state space.

```
Vector 'x.vct' (37/176 states)

Enter first state: 0
Enter last state: 10
0 0 0  : 8.7633885095484709e-01 (0)
0 0 1  : 1.4605647524579988e-02 (1)
0 0 2  : 1.4605647524579988e-02 (2)
0 0 3  : 2.4342745887546483e-04 (3)
0 0 4  : 1.4605647524579988e-02 (4)
0 0 5  : 2.4342745887546483e-04 (5)
0 0 6  : 2.4342745887546483e-04 (6)
0 0 8  : 2.4342745887546483e-04 (8)
0 0 9  : 2.4342745887546483e-04 (9)
0 0 10  : 2.4342745887546483e-04 (10)
Do you wish to save ALL reachable states in a file (y/n)? n

Do you wish to see (on screen) more elements (y/n)? n
```

Import a MARCA vector

This option allows the user to input a vector computed by MARCA (which may then be used, for example, to compare the results computed by PEPS and by MARCA).

```
Importing a vector from MARCA as solution of 'cnd/toto'

Enter MARCA vector file name: y.vct

Enter vector file name: z
 :-) file 'y.vct' read
 :-) file 'z.vct' saved
Vector 'y.vct' (37 states) imported and saved to 'z.vct
(176 remaining states from 176 original states)
```

4.2.5 Loading data structures

If chosen from the main menu, this option presents the following:

```
         ******* Load/Show/Remove Data Structures *******

 1) Show/Remove Structures           4) Load Aggregated Descriptor (agg)
 2) Load Textual Descriptor (dsc)    5) Load Probability Vector    (vct)
 3) Load Normalized Descriptor (cnd) 6) Load Sparse Matrix (hbf)
```

For each of these options (except the first), the user is asked for the file name (no extension, no path from the model directory).

4.2.6 Inspect data structures

This options is mainly useful for debugging purposes.

```
           ******* Inspect Data Structures *******

 1) Descriptor in Memory   (dsc or agg or cnd)
 2) Reachable State Space  (rss)
 3) Probability Vector     (vct)
 4) Sparse Matrix          (hbf)
1
There is no non-aggregated descriptor (dsc) in memory

Enter descriptor file name: toto
 :-) file 'dsc/toto.rss' read
 :-) file 'dsc/toto.ftb' read
 :-) file 'dsc/toto.dsc' read
 :-) file 'dsc/toto.dct' read
```

```
 :-) file 'dsc/toto.inf' read

-------------------------------------------------------------------------
Inspecting the descriptor dsc/toto.dsc
-------------------------------------------------------------------------
Product state space:               256 states
Reachable state space:             37 states
Number of automata:                8
Number of synchronized events:     0
-------------------------------------------------------------------------
Memory used by the descriptor:     1 Kbytes
Memory used by the function table: 0 Kbytes
Memory used by the rss vector:     0 Kbytes
Memory used by the results table:  0 Kbytes
All memory spent (all structures): 2 Kbytes
Automata sizes:
[ 2 2 2 2 2 2 2 2 ]
-------------------------------------------------------------------------

              ******* Inspect Data Structures *******

 1) Descriptor in Memory   (dsc or agg or cnd)
 2) Reachable State Space  (rss)
 3) Probability Vector     (vct)
 4) Sparse Matrix          (hbf)
2

-------------------------------------------------------------------------
Inspecting the reachable state space x.rss
-------------------------------------------------------------------------
Product state space:                  256 states
Informed Reachable state space:       37 states
Number of zero probability states:    219 states
Number of nonzero probability states: 37 states
-------------------------------------------------------------------------
Memory used by the rss vector: 0 Kbytes
Automata sizes:
[ 2 2 2 2 2 2 2 2 ]
-------------------------------------------------------------------------


              ******* Inspect Data Structures *******

 1) Descriptor in Memory   (dsc or agg or cnd)
 2) Reachable State Space  (rss)
 3) Probability Vector     (vct)
 4) Sparse Matrix          (hbf)
3

-------------------------------------------------------------------------
```

```
Inspecting the vector x.vct
--------------------------------------------------------------------------
Product state space:               256 states
Reachable state space:             37 states
Number of zero probability states: 219 states
--------------------------------------------------------------------------
Memory used by the vector: 3 Kbytes
Automata sizes:
[ 2 2 2 2 2 2 2 2 ]
--------------------------------------------------------------------------



              ******* Inspect Data Structures *******

 1) Descriptor in Memory   (dsc or agg or cnd)
 2) Reachable State Space  (rss)
 3) Probability Vector     (vct)
 4) Sparse Matrix          (hbf)
4

--------------------------------------------------------------------------
Inspecting the sparse matrix sparce.hbf
--------------------------------------------------------------------------
State space (matrix order):        37 states
Nonzero entries:                   0 elements
Memory used by the sparse matrix:  0 Kbytes
--------------------------------------------------------------------------
```

4.2.7 Sparse matrix facilities

Thus options is to permit the generation of sparse matrices.

```
         ******* Sparse matrix facilities *******

 1) generate a .hbf file from a SAN
 2) solve a SAN model using the sparse algorithm
 3) convert a .hbf file to/from MARCA format
 4) generate the state numbering of a SAN
```

If option 3) is chosen, we find:

```
Enter sparse matrix file name: m541
 :-) file 'm541.hbf' read
The matrix 'm541.hbf' is in the PEPS format (first column index is 0)
Converting to MARCA format (first row index is 1)
 :-) file 'MATX_hb' saved
```

The default name of the MARCA matrix is `MATX_hb` since this is the name required by this software package.

4.2.8 About this version

```
    PEPS - Performance Evaluation of Parallel Systems - version 2003

 PEPS is a software tool for solving Stochastic Automata Networks (SAN)

    The current version (2003) is an experimental version. Mathematical
basis for SANs may be obtained in the literature.  The main limitations
of this version are:
         - the function compilation procedure has size limitations;
         - the grouping procedure is not fully tested;
         - the saved files are large (ascii format);
         - only continuous-time models are accepted;
         - only numerical solutions are available
         - large functions may lead to memory problems

    PEPS is a collective software effort and its authors may be reached
at:
        peps@imag.fr or http://www-apache.imag.fr/software/peps/

    The distribution of this software is free,  but any changes in  the
source code is under the full responsability of the user. Any comercial
use of  PEPS  or modified versions of  PEPS cannot be performed without
the  formal autorization of the authors.  Any scientific benefit of the
use of the PEPS tool must mention it.

    This version of PEPS was released on: May 21 2003
             and it has been compiled on: Sep  1 2003 (optimized version)
```

4.2.9 Debugging models and PEPS

There are two options for compiling PEPS: one is optimized and useful for general model analysis; the other incorporates a number of tests and is useful for debugging PEPS itself (see section 4.1). Moreover, in the optimized version, inputting "d " under the main menu provides some means for printing and checking data.

```
  1) Compile a SAN model             5) Load data structures
  2) Solve a compiled SAN model      6) Inspect data structures
  3) Probability vector facilities   7) Sparse matrix facilities
  4) Preferences                     8) about this version

   0) exit PEPS (option 0 always allows to exit the current menu
d
```

```
       ******* Entering PEPS 2003 debug mode *******
            *** i.e., you shouldn't be here ***

 1) edit inner preferences   5) print rss
 2) print dsc                6) print cnd
 3) print dct                7) print cnd-diag
 4) print ftb                8) print inf
            9) Show Structures
```

4.2.10 Errors

If a fatal error occurs during the dialog(PEPS does not understand the answer or the answer is invalid), PEPS ask again the question (if the I/O stream does not fail).

If PEPS detects a potential error, it outputs a warning of the type:

```
** Warning: maybe not a probability vector (sum 7.8356589210131578e-01)! **
```

But the user may continue with the PEPS session.

Some errors may be detected which which arise from bugs in PEPS, from memory limits or bad file format (e.g. after a manual editing of a file generated by PEPS). These are

- for PEPS bugs, as error message of the type:

  ```
  Programming Error -(**** Invalid Index 2004  ****)
  (file 'autset.C' - line 297
  ****************************************************************
  ```

  or

  ```
  Programming Warning - (**** Invalid Index 2004  ****)
   (file 'autset.C' - line 297
   ****************************************************************
  ```

- memory limits

  ```
  ************* Function Evaluation Internal Error *************
  This function evaluation exceed the maximum stack size. You can
  change the function in order to keep the evaluation into the
  limit (20), or re-compile PEPS changing the constant 'max\_stack\_size'
  value in the file 'peps.h' (the maximum value needed by this function
  is: 25)
  ****************************************************************
  ```

  or

  ```
  ********************************************************************
  There is not enough virtual memory to run this operation in PEPS!
  ```

```
Sometimes this problem can be solved by removing the limits of your
shell. If your are using a UNIX system type 'man limit' to see how this
can de done. For example, you may try: 'unlimit vmemoryuse' and run
PEPS again.

In any case, this memory shortage is forcing PEPS to exit now!
*********************************************************************
```

- bad file format

```
Descriptor Compilation Error - Automata number mismatch
*****************************************************************
```

or

```
Bad Function Definition - (F12)
*****************************************************************
```

or

```
Bad Function Semantic Definition - (F18)
*****************************************************************
```

Bibliography

[1] M.Ajmone-Marsan, G.Balbo, G.Conte, S.Donatelli, G.Franceschinis. *Modeling 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, L. Brenner, P. Fernandes, B.Plateau and W.J. Stewart. The PEPS Software Tool. In: 4^{th} *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Urbana, Illinois, USA, 2003. Springer-Verlag.

[4] A. Benoit, B. Plateau and W.J. Stewart. Memory-efficient Kronecker algorithms with applications to the modeling of parallel systems. *To appear in PMEO-PDS'03*, 2003.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] E.Gelenbe, G.Pujolle. *Introduction to Queueing Networks*. John Wiley, 1997.

[11] J.Hillston. *A Compositional Approach for Performance Modeling*. Ph.D. Thesis, University of Edinburg, United Kingdom, 1994.

[12] 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.

[13] B.Plateau. *De l'Evaluation du Parellélisme et de la Synchronisation*. Thèse de Doctorat d'Etat, Paris-Sud, Orsay, France, 1984.

[14] 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.

[15] B.Plateau, J.M.Fourneau, K.Lee. PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In: R.Puigjaner, D.Potier, eds. *Modeling Techniques and Tools for Computer Performance Evaluation*, 1988.

[16] PEPS team. PEPS 2003 *Software Tool.* On-line document available at `http://www-apache.imag.fr/software/peps`, visited Feb. 14th, 2003.

[17] Y.Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, 1995.

[18] 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.

[19] W.J.Stewart. MARCA: Markov Chain Analyzer. *IEEE Computer Repository* No. R76 232, 1976.

[20] W.J.Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[21] Sun Microsystems *The JIT Compiler Interface Specification.* On-line document available at `http://java.sun.com/docs/jit_interface.html`, visited Feb. 14th, 2003.