PEPS 2007 User manual

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

The PEPS 2007 User manual author
Leonardo Brenner

November 10, 2008

Contents

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) [**?**, **?**] 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 [**?**], generalized stochastic Petri nets [**?**], stochastic reward nets [**?**] and stochastic activity nets [**?**] 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 [**?**] and process algebras [**?**].

The basic idea is to represent the matrix of the Markov chain by means of a tensor (Kronecker) formula, called *descriptor* [**?**]. 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 [**?**], 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 [**?**]. 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.

Chapter 2

SAN Presentation

In a SAN [?, ?] 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 [?].

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 $i = 1 \ldots N$) 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 $\tilde{x} = (x^{(1)}, \ldots, x^{(N)})$. $\mathcal{S} = \mathcal{S}^{(1)} \times \cdots \times \mathcal{S}^{(N)}$ 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 **??** 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 $1 - \pi$.

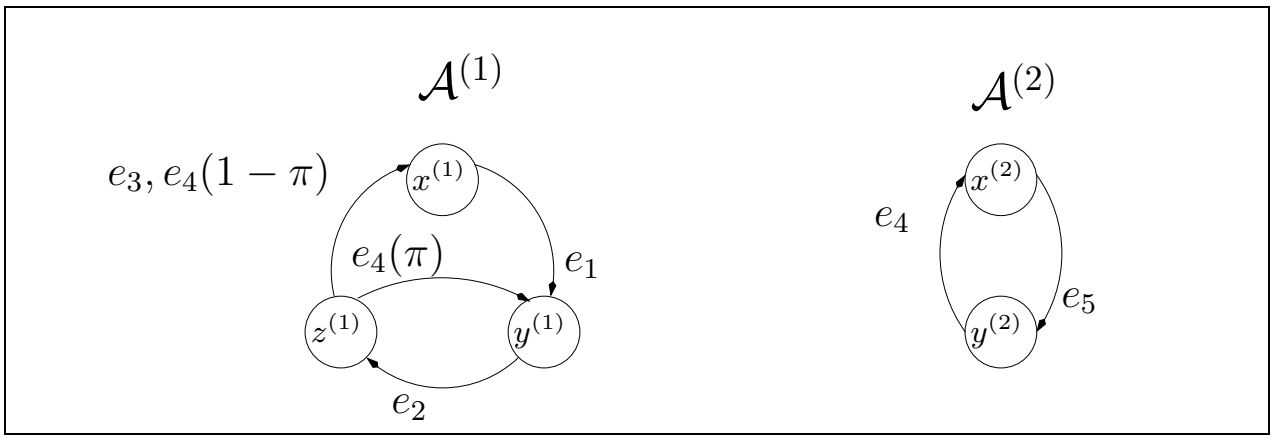

Figure 2.1: Very Simple SAN model example

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.

Chapter 3

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:

- Textual description of continuous-time SAN models (without replicas);
- Stationary solution of models using Arnoldi, GMRES and Power iterative methods [**?**, **?**];
- Numerical optimization regarding functional dependencies, diagonal pre-computation, pre-conditioning and algebraic aggregation of automata [**?**]; 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 2007 is able to:

- Read a file which describes a SAN in a textual format, replicas features are improved. This interface is described in Chapter **??**. Section **??** provides simple examples of models that can be used to experiment with the software.
- 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. Theses features are described in Chapter **??**
- Provide the user with a selection of iterative solution methods to compute stationary and transient probability vector and a number of numerical options, as it is explained in Chapter **??**.

- Inspect data structures such as the reachable state space, the stationary probability vector (and compare two of them), and the SAN descriptor.

- Output results and data structure files: results, probability vectors, SAN descriptor, HBF format of the descriptor compatible with the software MARCA [**?**].

PEPS 2007 is composed by some modules. Each module implements a step to a model resolution. The modules are grouped in three phases: *Description*, *Compiling*, and *Solution*. Description phase is composed by the interface modules. The data structures modules compose the compiling phase and the solution phase concerns the solution modules.

The next sections presents a brief description of the steps from to describe a model til to solve its in the PEPS software. Including PEPS installation procedures.

3.1 PEPS 2007 Installation

Being a academic open-source software, the PEPS 2007 installation is quite simple and do not demand any special feature from your system. The *Makefile* does 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. PEPS 2007 uses the compilation of functional elements *just-in-time* [**?**]. 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).

3.1.1 Getting the source

Retrieve from our site the file `Peps2007.tgz`, for full PEPS 2007 package or the compressed file for each individual module. Use `tar -xvzf file` (*e.g.* tar -xvzf Peps2007.tgz for full PEPS 2007 package) to uncompress this file and to generate the directory `Peps2007` and all the hierachy of files.

3.1.2 Compiling

If you want to compile all PEPS 2007 modules, just type `make` and the binary files of PEPS will be generated. You can compile just one individual module. In this case type `make module_name` (*e.g.* `make compile_san`) or run `make` into module directory. You can change the compiling options for each module into module directory.

3.1.3 Binary files

In the directory `./bin/` in each module directory there is a file with the module name (*e.g.* compile_san) which is the executable. To run a Peps module just type the module name. If you compiled the full PEPS 2007 package, all executable files generated by each module are copied to the directory `./peps2007/bin/`. The first time that Peps is running in a directory, it will initialize all auxiliary subdirectories.

3.2 Model Description

A model must be described in a simple text file and the file name must have `.san` as extention. You can use a simple text file editor like *vi* to create your file model.

The model description must respect the SAN interface, described in Chapter **??**. Some models examples are presented in Section **??**.

3.3 Compiling a model

The first step to solve a SAN model is to compile the `.san` model file to generate the Full Markovian Descriptor files. To compile a `.san` file, we use *compile_san* module as follow:

Command: `compile_san file`

Typical module output:

```
./compile_san rs
Start model compilation
First Passage
Compiling identfier block
Compiling event block
Compiling reachability function
Compiling network block
Compiling results block
Creating automata and states structures
Second Passage
Compiling identfier block
Compiling event block
Compiling reachability function
Writing events informations
Compiling network block
Compiling results block
Checking events integrity
Model compiled
Creating description files model
 :-) file 'des/rs.des' saved
 :-) file 'des/rs.dic' saved
 :-) file 'des/rs.fct' saved
 :-) file 'des/rs.tft' saved
 :-) file 'des/rs.res' saved
The description files to rs model were created in "des" directory.
```

The second step in the compiling phase is to transform the Full Markovian Descriptor files in Sparse Markovian Descriptor. This step keeps only non-zero values of the matrices and run the aggregation procedures, if choice. To perform this step, we use the *compile_dsc* module:

Command: `compile_dsc [options] file`

Typical using example with standard options:

```
./compile_dsc rs

Compilation of a SAN model (Internal Descriptor Generation)
Compile_Network
Compile_Function_Table
 :-) file 'des/rs.tft' read
 :-) file 'dsc/rs.ftb' saved
Compile_Descriptor
 :-) file 'des/rs.des' read
 :-) file 'dsc/rs.dsc' saved
Compile_Reacheable_SS
 :-) file 'dsc/rs.rss' saved
 :-) file 'des/rs.fct' read
Compile_Dictionary
 :-) file 'dsc/rs.dct' saved
 :-) file 'des/rs.dic' read
Compile_Integration_Function
 :-) file 'des/rs.res' read
 :-) file 'dsc/rs.inf' saved

Translation performed:  compilation of a SAN model
 - user time spent:     4.0000000000000001e-03 seconds
 - system time spent:   0.0000000000000000e+00 seconds
 - real time spent:     2.7468191855587065e-01 seconds

Thanks for using PEPS!
```

The last step in the compiling phase is the model normalization. This step normalizes the Sparse Markovian Descriptor files to Continuous Normalized Descriptor. Two modules can be used in this step. The *norm_dsc_ex* module uses an extended vector format and *norm_dsc_sp* module uses a sparse vector format.

Command: `norm_dsc_ex file` or `norm_dsc_sp file`

Typical using example output message:

```
./norm_dsc_ex rs

Normalization of a SAN Descriptor
 :-) file 'dsc/rs.rss' read
 :-) file 'dsc/rs.ftb' read
 :-) file 'dsc/rs.dsc' read
 :-) file 'dsc/rs.dct' read
 :-) file 'cnd/rs.cnd' saved
 :-) file 'cnd/rs.ftb' saved
 :-) file 'cnd/rs.rss' saved
 :-) file 'peps/peps2007/bin/jit/peps_jit.C' saved
```

```
Translation performed:  normalization of a SAN descriptor
  (largest element in reachable states: 1.5000000000000000e+01)
 - user time spent:     4.0000000000000001e-03 seconds
 - system time spent:   4.0000000000000001e-03 seconds
 - real time spent:     8.9477301982697099e-01 seconds

Thanks for using PEPS!
```

3.4 Solving a model

After the compiling phase, the model can be solved. The solution methods are implemented in two modules, and as in the last compiling step, *solve_cnd_ex* module uses an extended vector format and *solve_cnd_sp* module uses a sparse vector format. If you compile your model using the extended vector, we must use the extended vector representation also to solve its.

Command:`solve_cnd_ex [options] file` or `solve_cnd_sp [options] file`

Typical using example output message:

```
./solve_cnd_ex rs
 :-) file 'cnd/rs.rss' read
 :-) file 'cnd/rs.ftb' read
 :-) file 'peps/peps2007/bin/jit/peps_jit.C' saved
 :-) file 'cnd/rs.cnd' read
 :-) file 'dsc/rs.dct' read

Solution of the model 'cnd/rs.cnd' (4 automata - 11/16 states)

Enter vector file name: v
Iteration 0: largest: 1.6363636363636364e-01 (0) smallest: 5.4545454545454550e-02 (3)
Iteration 10: largest: 2.1944567435636364e-01 (0) smallest: 5.4861419985454539e-02 (3)
Iteration 20: largest: 2.2192409302507105e-01 (0) smallest: 5.5481023256267900e-02 (3)
Iteration 30: largest: 2.2219021084342858e-01 (0) smallest: 5.5547552710857144e-02 (3)
Iteration 40: largest: 2.2221878502659678e-01 (0) smallest: 5.5554696256649189e-02 (3)
Iteration 50: largest: 2.2222185315615217e-01 (0) smallest: 5.5555463289038043e-02 (3)
Iteration 60: largest: 2.2222218259405468e-01 (0) smallest: 5.5555545648513671e-02 (3)
Iteration 70: largest: 2.2222221796718014e-01 (0) smallest: 5.5555554491795035e-02 (3)
Iteration 80: largest: 2.2222222176534054e-01 (0) smallest: 5.5555555441335135e-02 (3)
Iteration 90: largest: 2.2222222217316495e-01 (0) smallest: 5.5555555543291231e-02 (3)
Iteration 91:
Power solution
Number of iterations:   92
 - user time spent:     -8.3432745304548583e-20 seconds
 - system time spent:   -2.4987471944001860e-19 seconds
 - real time spent:     3.4348982153460383e-03 seconds
Residual Error: 7.0642436345025317e-11 - The method converged (solution found)!
 :-) file 'v.vct' saved
```

```
 :-) file 'rs.tim' saved

Thanks for using PEPS!
```

Part I

PEPS BASIC MODULES

Chapter 4

Interface

This chapter presents the SAN textual interface to PEPS 2007. This new textual interface is full compatible with PEPS 2003 and incorporates new features to replications sets. Addictionally, we present some modeling examples to show the interface powerful to model systems based replicated components.

4.1 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 2007 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 2007 textual formalism used to describe SAN models. To be compatible with PEPS 2007 , any file describing a SAN should have the suffix `.san`. Figure **??** shows an overview of the PEPS input structure. A SAN description is composed of five blocks (Figure **??**) 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.

[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
  $loc < evt\_name > (< rate >)$
  $syn < evt\_name > (< rate >)$
  // with replication
  $loc < evt\_name > [replication\_domain](< rate >)$
  $syn < evt\_name > [replication\_domain](< rate >)$

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

network $< net\_name > (< type >)$
  $aut < aut\_name > \{[replication\_domain]\}$
    $stt < stt\_name > \{[replication\_domain]\}\ \{(reward)\}$
      $to(< stt\_name > \{[replication\_domain]\}\ \{/f\_cond\}\ )$
        $< evt\_name > \{[replication\_domain]\}\ \{(< prob >)\}\ \{/f\_cond\}$
        ...
        $< evt\_name > \{[replication\_domain]\}\ \{(< prob >)\}\ \{/f\_cond\}$
      ...
    $from < stt\_name >$
      $to(< stt\_name > \{[replication\_domain]\}\ \{/f\_cond\}\ )$
        $< evt\_name > \{[replication\_domain]\}\ \{(< prob >)\}\ \{/f\_cond\}$
    ...
    $stt < stt\_name > \{[replication\_domain]\}\ \{(reward)\}$
      $to(< stt\_name > \{[replication\_domain]\}\ \{/f\_cond\}\ )$
        $< evt\_name > \{[replication\_domain]\}\ \{(< prob >)\}\ \{/f\_cond\}$
  ...
  $aut < aut\_name > \{[replication\_domain]\}$
    ...

results
  $< res\_name >=< exp >$ **;**

Figure 4.1: Modular structure of SAN textual format

4.1.1 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 states identifiers. In this last case, the expressions are functions defined on the SAN model state space. For example, "the number of automata in state $n0$" (which gives an integer result) can be expressed as "$nb\ n0$". A function that returns the value 4 if two automata ($A1$ and $A2$) are in different states, and the value 0 otherwise, is expressed as "$(st\ A1\ !=\ st\ A2)\ *\ 4$". Comparison operators return the value "1" for a true result and the value "0" for a false result. The format of an expression is described in Section **??**.

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]". Identifiers can be used to define an interval "[1..ID1,5,7..ID2]", where ID1 and ID2 are identifiers with constant values. In all cases, domain must respect an increasing value order;
- "$< exp >$" is a real number or a mathematical expression. The mathematical expressions are described in Section **??**. 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 $A[10]$.

4.1.2 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).

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
$loc < evt\_name > [replication\_domain](< rate >)$
$syn < evt\_name > [replication\_domain](< rate >)$
...

- "*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;
- "[*replication_domain*]" is a set of indexes. The *replication_domain* must be a domain identifier defined in **identifiers** block. An event can be replicated til three levels. Each level is defined by [*replication_domain*]. For example, an event replicated in two levels is defined as $<$ *evt_name* $>$ [*replication_domain*][*replication_domain*];
- "$<$ *rate* $>$" defines the rate of the event. It must be an expression identifier declared in the **identifiers** block.

4.1.3 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 2007.

Format of the **reachability** block

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

4.1.4 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.

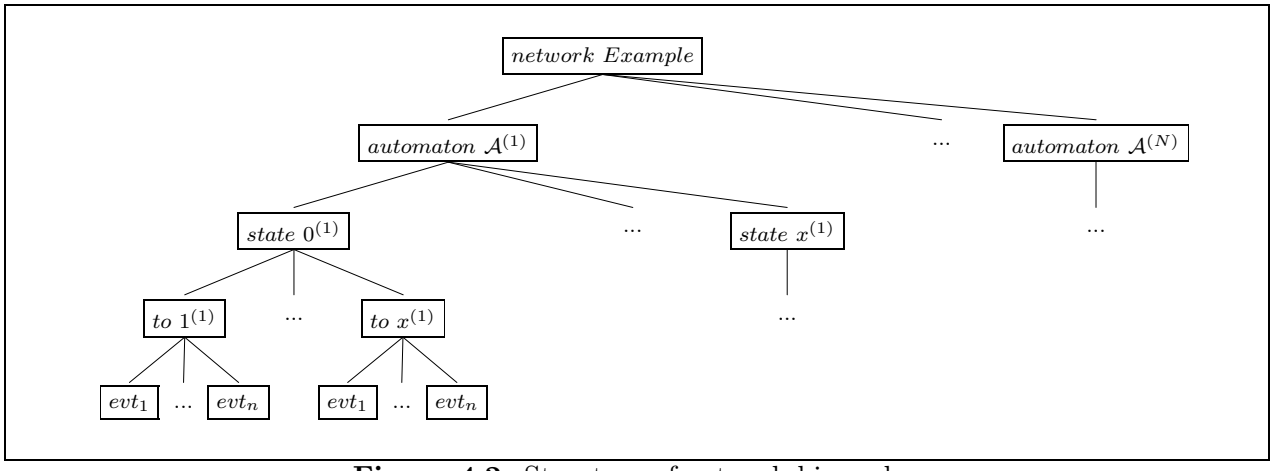

Figure 4.2: Structure of network hierarchy.

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* $>$ {[*replication_domain*]}
    stt $<$ *stt_name* $>$ {[*replication_domain*]} {(*reward*)}
      to($<$ *stt_name* $>$ {[*replication_domain*]} {/*f_cond*} *)*
        $<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}
  ... // *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 2007.

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

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* $>$ {[*replication_domain*]}
    stt $<$ *stt_name* $>$ {[*replication_domain*]} {(*reward*)}
      to($<$ *stt_name* $>$ {[*replication_domain*]} {/*f_cond*} *)*
        $<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}
  ... // *States description*

- "$<$ *aut_name* $>$" is the name of the automaton. It is an alphanumeric identifier and it may be used for expression definitions;
- "[*replication_domain*]" is a set of indexes. The *replication_domain* is a domain identifier. This identifier must defined in **identifiers** block.

State Description

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

Format of the *stt* block

stt $<$ *stt_name* $>$ {[*replication_domain*]} {(*reward*)}
    to($<$ *stt_name* $>$ {[*replication_domain*]} {/*f_cond*} *)*
      $<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}
  ... // *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;
- "[*replication_domain*]" is the number of times that the state appears in the automaton. It is described 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* $>$ {[*replication_domain*]} {/*f_cond*} *)*
    $<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}
  ... // *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" must be an identifier previously defined in **identifiers** block.

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 or a set of replicated states. In this case, three situation are possibles:

- use a constant to define a static state;
- use a function to define one variable state, where the target state index is defined by the current state index;
- use a domain to define multiply transitions.

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* $>$ {[*replication_domain*]} {/*f_cond*} *)*
$<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}
... // *Events description*

- "/*f_cond*" defines a condition to include the transition. *f_cond* is an function identifier previously defined in **identifiers** block. Normally, this function value depend of the current automaton and/or current and/or target state.

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, and firing condition.

Format of the *event* block

$<$ *evt_name* $>$ {[*replication_domain*]} {($<$ *prob* $>$)} {/*f_cond*}

- "$<$ *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 or a set of replicated states. In this case, three situation are possibles:
  - use a constant to define a static state;
  - use a function to define one variable event index, where the target event index is defined by the current automaton/state index or the target state;
  - use a domain to define multiply transitions.

  Remember, we can have three replications level for each event, the three situations described here are able in each replication level. To address events replicated in two or three levels you must use "[x][x]" and "[x][x][x]", respectively;
- "($<$ *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;
- "/*f_cond*" defines a condition to include an event. *f_cond* is an function identifier previously defined in **identifiers** block. Normally, this function value depend of the current automaton and/or current and/or target state.

4.1.5 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 **??**.

4.1.6 Function 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 automata or states identifiers.

The format of the operators is summarized in table **??**. 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 "+", "−", "∗", "/", "*div*", "*mod*", "*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. We can have two kind of functions. The first one is called "description functions" and the second one is "SAN functions".

Description Functions

Description functions are used in the model description and they are useful to replicate states, transitions, and events. When Peps 2007 genere the Markovian Descriptor (set of matrices that describe the model), it take in to account these functions to build the model.

Peps 2007 defines three functions:

- **at** : returns the current automaton index. If automaton is a replicated automaton then this function returns the internal replication index else it returns the general automaton index in the model;
- **sts** : gives the current state index in the current automaton. As well the previous functions, this functions returns the internal replication index to a replicated state and the general state index to a non replicated state;
- **std** : this functions returns the target state in a transition. The states indexes returned by this functions follow the previous functions behaviour.

SAN Functions

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
Description Operators		
at	gives the current automaton index	at
sts	gives the current state index	sts
std	gives the target state index	std
SAN Operators		
st < *aut_name* > **@** < *aut_name* >	gives the current state of automaton "< *aut_name* >" the same as above, another notation	st processA @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		
< *exp*1 > + < *exp*2 >	sum of "< *exp*1 >" and "< *exp*2 >"	5 + 3
< *exp*1 > − < *exp*2 >	substraction of "< *exp*2 >"minus "< *exp*1 >"	5 − 3
< *exp*1 > ∗ < *exp*2 >	product of "< *exp*1 >" and "< *exp*2 >"	5 ∗ 3
< *exp*1 > / < *exp*2 >	division of "< *exp*1 >" by "< *exp*2 >"	5 / 3
< *exp*1 > *div* < *exp*2 >	integer division of "< *exp*1 >" and "< *exp*2 >"	5 *div* 3
< *exp*1 > *mod* < *exp*2 >	rest of integer division of "< *exp*1 >" by "< *exp*2 >"	5 *mod* 3
min (< *exp*1 >, < *exp*2 >]	gives the miniimum of two arguments	min(5,3)
max (< *exp*1 >, < *exp*2 >]	gives the maximum of two arguments	max(5,3)
Relational Operators		
< *exp*1 > == < *exp*2 >	if "< *exp*1 >" is equal to "< *exp*2 >", gives true ("1") else false ("0")	5 == 3
< *exp*1 > ! = < *exp*2 >	if "< *exp*1 >" is not equal to "< *exp*2 >", givess true ("1") else false ("0")	5 ! = 3
< *exp*1 > < < *exp*2 >	if "< *exp*1 >" is smaller than "< *exp*2 >", gives true ("1") else false ("0")	5 < 3
< *exp*1 > <= < *exp*2 >	if "< *exp*1 >" is smaller or equal than "< *exp*2 >", gives true ("1") else false ("0")	5 <= 3
< *exp*1 > > < *exp*2 >	if "< *exp*1 >" is greather than "< *exp*2 >", results true ("1") else results false ("0")	5 > 3
< *exp*1 > >= < *exp*2 >	if "< *exp*1 >" is greater or equal than "< *exp*2 >", gives true ("1") else false ("0")	5 >= 3
Logical Operators		
! < *exp* >	logical "not"	5 > 3) ! (7 < 8)
< *exp*1 > || < *exp*2 >	logical "or"	(5 > 3) || (7 < 8)
< *exp*1 > && < *exp*2 >	logical "and"	(5 > 3) && (7 < 8)
< *exp*1 > ∧ < *exp*2 >	logical "XOR"	(5 > 3) ∧ (7 < 8)

Table 4.1: Operators

4.2 Modeling Examples

In this section three examples are presented to illustrate the modeling power and the computational effectiveness of PEPS 2007. For each example, the generic SAN model is described.

4.2.1 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.

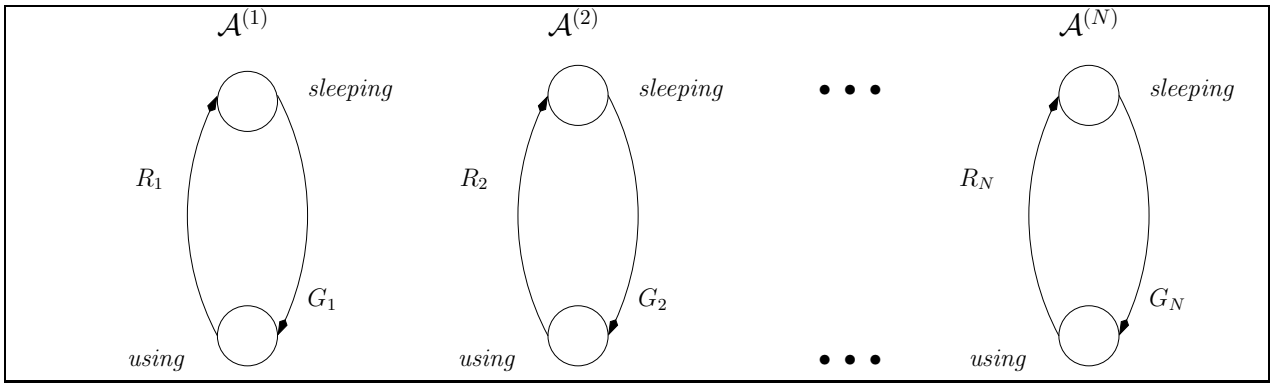

Figure 4.3: Resource Sharing Model - version 1

In our SAN representation (Figure **??**), 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 **??** 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)  // local event G1 has rate f1
   loc R1 (mu1) // local event R1 has rate mu1
   loc G2 (f2)  // local event G2 has rate f2
   loc R2 (mu2) // local event R2 has rate mu2
   loc G3 (f3)  // local event G3 has rate f3
   loc R3 (mu3) // local event R3 has rate mu3
   loc G4 (f4)  // local event G4 has rate f4
   loc R4 (mu4) // local event R4 has rate mu4

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       = [0..3];  // 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 processes
   i       = at;      // current automaton index
   f       = lambda * (nb using < R);

events
   loc Acq[N] (f)   // local events Acq have rate f
   loc Rel[N] (mu)  // local events Rel have rate mu

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)    Acq[i]
    stt using
      to(sleeping) Rel[i]

results
  full     = nb using == R;     // probability of all resources being used
  empty    = nb using == 0;     // probability of all resources being available
  use1     = st P[0] == 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 **??** 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).

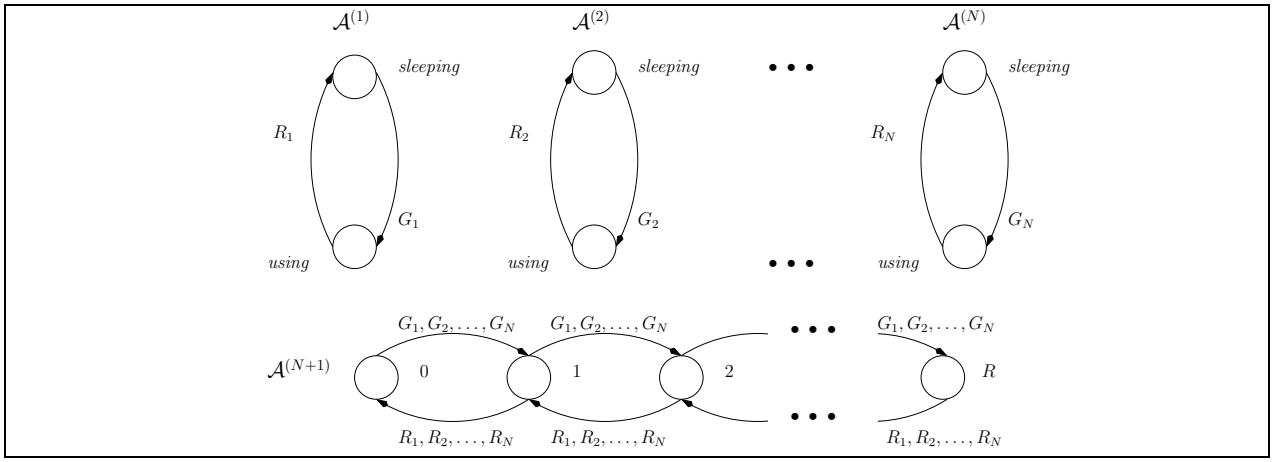

Figure 4.4: Resource Sharing Model without functions - version 2

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

```
//======================================== RS model version 2 ======================================
//                                          N=4, R=2
//==================================================================================================
```

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

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
```

4.2.2 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 **??**. 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.

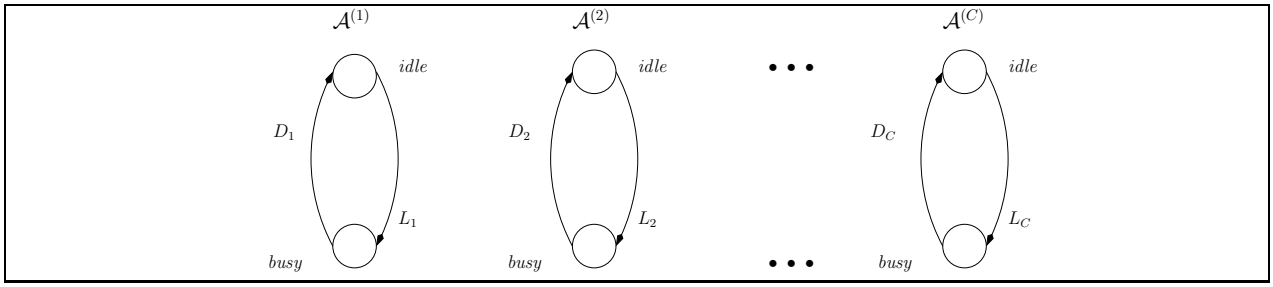

Figure 4.5: First Server Available Model

The Peps 2007 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)
   loc D1 (mu1)
   loc L2 (f2)
   loc D2 (mu2)
   loc L3 (f3)
   loc D3 (mu3)
   loc L4 (f4)
   loc D4 (mu4)

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;
```

4.2.3 Example: A Mixed Queueing Network

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

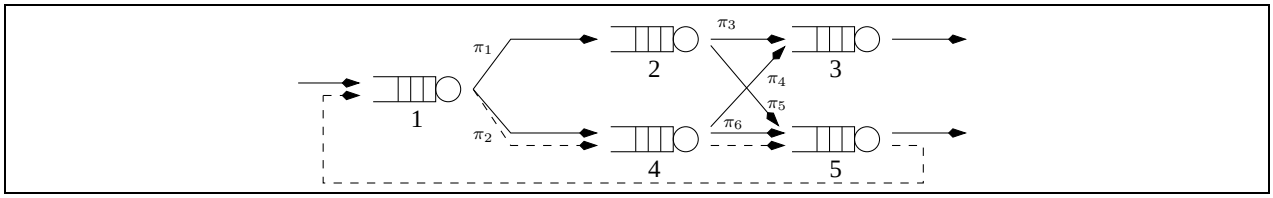

Figure 4.6: Mixed queueing network

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 [**?**].

```
//================================================ 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 5

Data Structure

The data structures used in Peps 2007 can be divided in two large blocks. The first block uses a Kronecker (tensor) format and the second one uses a sparse matrix format (HBF). Each block and theirs modules implementation are described below.

5.1 Kronecker Format

In Kronecker format, we use a set of small matrices to store a large model. This structured format allows to save storage space in comparison to non-structured format. Compiling phase for kronecker structures is composed by two step. The first one transforms a Full Markovian Descriptor in a Sparse Markovian Descriptor and the second one transforms a Sparse Markovian Descriptor in a Continuous Normalized Descriptor. Theses steps are described below.

5.1.1 Sparse Markovian Descriptor

The first step is to convert the set of small matrices (Full Markovian Descriptor) created in SANcompilation module in a set of sparse matrices (Sparse Markovian Descriptor). This procedure converts the textual matrices format used in descrption phase to a Peps 2007 internal representation format.

This step also implements the aggregation methods. Two aggregation methods are implemented in Peps 2007 . The first one does an algebrique automata aggregation, *i.e.* tensor operation are applied in a set of automata. The second method applies a replicas aggregation methods. More informations about theses techniques can be find in [**?**].

This step is implemented in compile_dsc module.

compile_dsc Module Implementation

This module implements the Markovian Descriptor generation in the internal Peps format from the standard input (textual matrices generated in .san compilation phase).

Source code path: `peps2007/src/ds/compile_dsc`
Required files: `.des, .dic, .fct, .tft, .res`
Generated files: `.dsc, .dct, .ftb, .inf, .rss`

Command: `compile_dsc [options] file`

Options:

aggr This option applies the algebrique aggregation method.

lump This option is used to apply the semantic aggregation method.

5.1.2 Normalized Markovian Descriptor

The second data manipulation step normalizes the model and extract the matrices diagonal elements. The diagonal generation removes all synchronizing adjust matrices and all diagonal elements in local matrices. All theses elements will be placed in a diagonal vector used by the solutions methods. This technique improves the solution methods performance.

In this step, some model analisys are performed, as product and reachable space state analysis, constant functions replacement, matrices multiplication order generation, etc.

Two modules was implemented to perform this step. The first one uses a extended vector implementation (norm_dsc_ex) and the second one uses a sparse vector implementation to store the reachable space state.

norm_dsc_ex Module Implementation

This module implements the Markovian Description normalization using a extended vector.

Source code path: `peps2007/src/ds/norm_dsc_ex`
Required files: `.dsc, .dct, .ftb, .inf, .rss`
Generated files: `.cnd, .ftb, .rss`
Command: `norm_dsc_ex file`

norm_dsc_sp Module Implementation

This module implements the Markovian Description normalization using a sparse vector.

Source code path: `peps2007/src/ds/norm_dsc_sp`
Required files: `.dsc, .dct, .ftb, .inf, .rss`
Generated files: `.cnd, .ftb, .rss`
Command: `norm_dsc_sp file`

5.2 Harwell-Boeing Format (HBF)

5.2.1 HBF Generation

This module performs the equivalent Markov Chain generation from the SAN model. This Markov Chain represented by a transition matrix is stored in HBF format.

This generation is implemented in gen_hbf module..

gen_hbf Module Implementation

This module implements the equivalent markovian model (in hbf format) generator from the SAN Markovian Descriptor.

Source code path: `peps2007/src/ds/gen_hbf`
Required files: `.dsc, .dct, .ftb, .inf, .rss`
Generated files: `.hbf`
Command: `gen_hbf [option] file`

Option:

conv This option converte the sparse matrix generated by PEPS (C language) in a sparse matrix in MARCA pattern (Fortran language).

Chapter 6

Solution Methods

Peps 2007 implements two sets of methods to analytical solutions of a SAN model. The first set works with a kronecker format and the second set works with a sparse format (HBF).

6.1 Kronecker Format

6.1.1 Shuffle

The shuffle method basic idea is multiply the probability vector by the set of matrices that compose the Markovian Descriptor. In fact, each small matrix multiplies a part of the vector. The sum of all small matrices multiplication is a complete vector-descriptor multiplication.

This multiplication method was implemented in two versions. The first one works with extended vector (with the global espace state size). The second one works with sparse vector (only the reachable space state size).

solve_cnd_ex Module Implementation

This module implements the solution methods using an extended vector.

Source code path: `peps2007/src/solve/solve_cnd_ex`
Required files: `.cnd, .ftb, .rss, .inf, .dct`
Generated files: `.vct, .tim`
Command:`solve_cnd_ex [options] file`

solve_cnd_sp Module Implementation

This module implements the solution methods using a sparse vector.

Source code path: `peps2007/src/solve/solve_cnd_sp`
Required files: `.cnd, .ftb, .rss, .inf, .dct`
Generated files: `.vct, .tim`
Command:`solve_cnd_sp [options] file`

These modules have almost the same options and we will explaine only one time. The options that can be used for just one module will be mentioned.

Options:

-sol_meth *method* This option set solution method that will be used to solve the model. Four methods are proposed:

power use the power method; Power method is the default option.

gmres use the gmres method. This method is not implemented to solve_cnd_sp module;

arnoldi use the arnoldi method. This method is not implemented to solve_cnd_sp module;

unifor use the uniformization method.

-int_func Set the function integration to true. This option will integrate the result functions.

-iter n Set the maximum number of iteration to n. n default value is 1000.

-min_err *err* Set the tolerance accepted to *err*. *err* default value is 1.0e-10.

-iter_type *type* This option set the stop iteration criterion. Three type are proposed:

fix set stop criterion to fixed iteration number;

stb set stop criterion to stability test;

cnv set stop criterion to convergence test. The convergence test is the default option.

-err_type This option set the error test. Three type are proposed:

abs_ind set the error test to maximum of individual absolute errors. The maximum of individual absolute errors is the default option;

abs_acc set the error test to accumulated individual absolute error;

rel_ind set the error test to maximum individual relative error.

-vec_type Set the initial vector type. We have three possibilities:

eq set the initial vector to equiprobable. Default option;

in read a user vector as initial vector. The vector is read from a file;

ap use an aproximated by the inverse of the diagonal vector.

-meth_type *type* This option set the Vector-Descriptor product method. PEPS implement 4 methods:

A use avoid permutation method;

B use minimize permutations;

C use minimize function evaluations;

M use mixed method. Default option.

-precond This option set the diagonal preconditioning as true.

-zin_impl *type* Set the zin representation format to *type*. This option is implemented only to solve_cnd_sp module and two zin types are proposed:

full use full implementation (including zeros);

sparse use sparse implementation (only non-zeros).

-krylov n Set the krylov subspace size to n. Default krylov subspace size is 10. This option is not implemented to solve_cnd_sp module.

6.2 Harwell-Boeing Format (HBF)

This section presents solution methods based in sparse format (HBF).

6.2.1 HBF

This module implements the vector infinitesimal generator multiplication using a sparse matrix representation for the infinitesimal generator.

solve_hbf Module Implementation

Source code path: `peps2007/src/solve/solve_hbf`
Required files: `.hbf`
Generated files: `.vct`
Command:`solve_hbf [options] file`

Options:

-iter n Set the maximum number of iteration to n. n default value is 1000.

-min_err *err* Set the tolerance accepted to *err*. *err* default value is 1.0e-10.

-iter_type *type* This option set the stop iteration criterion. Three type are proposed:

fix set stop criterion to fixed iteration number;

stb set stop criterion to stability test;

cnv set stop criterion to convergence test. The convergence test is the default option.

-err_type This option set the error test. Three type are proposed:

abs_ind set the error test to maximum of individual absolute errors. The maximum of individual absolute errors is the default option;

abs_acc set the error test to accumulated individual absolute error;

rel_ind set the error test to maximum individual relative error.

-vec_type Set the initial vector type. We have three possibilities:

eq set the initial vector to equiprobable. Default option;

in read a user vector as initial vector. The vector is read from a file;

ap use an aproximated by the inverse of the diagonal vector.

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.