Manual for the SPNP Package
Version 3.1

Gianfranco Ciardo
Jogesh K. Muppala

October 18, 1992

Contact: Kishor S. Trivedi
Department of Electrical Engineering
Duke University
Durham, NC – 27706.
(919) - 660 - 5269
(919) - 493 - 6563
kst@egr.duke.edu

1 Introduction

The Stochastic Petri Net Package (SPNP) is a versatile modeling tool for solution of Stochastic Petri Nets (SPN) models. The SPN models are described in the input language for SPNP called **CSPL** (C-based SPN Language). The CSPL is an extension of the **C** programming language [17] with additional constructs which facilitate easy description of SPN models. The full power and generality of C is available, but a working knowledge of C is sufficient to use SPNP effectively.

The SPN models specified to SPNP are actually "SPN Reward Models" or Stochastic Reward Nets (SRNs) [5, 6] which are based on the "Markov Reward Models" paradigm [11, 27]. This provides a powerful modeling environment for:

- Dependability (Reliability,Availability,Safety) analysis.
- Performance Analysis.
- Performability modeling.

A number of important Petri net constructs like marking dependency, variable cardinality arc and enabling functions [5] facilitate the construction of models for complex systems. The package also allows logical analysis on the Petri net whereby any general assertions defined on the Petri net are checked for each marking of the net. The SRN may be solved to obtain either steady-state metrics or transient metrics. The package allows the specification of custom measures although a standard set of measures are available. The measures are defined in terms of rewards associated with the markings of the SRN. Sensitivity analysis allows the user to evaluate the effect of changes in an input parameter on the output measures. This is useful in system optimization and bottleneck analysis.

This manual describes Version 3.1 of SPNP (Stochastic Petri Net Package), running under the UNIX system on a wide array of platforms (VAX, Sun 3 and 4, Convex, Gould, NeXT, CRAY), AIX system (RS/6000), OS/2 system (PS/2), and VMS system (VAX). The description will apply mainly to UNIX-based systems; the appendix contains some minor differences for the VMS version.

A basic knowledge of the stochastic Petri net (SPN) formalism and Markov chains is assumed. The reader should consult [24, 25] if unfamiliar with some Petri net (PN) concepts. The SPN model we adopt is best described in [5, 6], but it may be useful to consult [2]. For a reference on the C language, see [17]. For further information on Markov chains, performance modeling and reliability modeling see [29] while for performability modeling see [27, 30]. Markov and Markov reward model solution techniques are surveyed in [26]. Sensitivity analysis of Markov and Markov reward models is discussed in [3] and the senisitivity analysis of SRN models is discussed in [19]. Several papers have appeared in the literature where SPNP was used [4, 7, 8, 9, 13, 12, 14, 15, 16, 20, 23, 22, 28, 18, 1]. A 6-hour long VHS tape

for a course on "Putting Stochastic Petri Nets to Work," by K. Trivedi and G. Ciardo can be ordered from USC-ITV by calling (213)-740-0119.

Although model hierarchies are not built into SPNP, hierarchical SRN models can be exercised using a UNIX shell file (VMS .com file) and submodels can communicate information via files that can be declared and opened inside individual submodel SPNP input files. Examples of papers using model hierarchies and fixed-point iteration include [4, 7, 9, 16, 20, 22, 23, 28]. In the next section, we review the SRN terminology. Then we present the structure of the package and we explain how to describe a SRN to the package. The format of the files generated by the package is then presented, followed by the list of the available options. In the conclusion we list future work planned on the package.

2 Terminology

The concept of *bag* will be used in the following, so we describe it here. For a complete definition and examples see [25]. The concept of bag extends the one of set. If $\mathbf{x}$ is an element of the set $\mathbf{S}$, then $\mathbf{S} \cup \{\mathbf{x}\} = \mathbf{S}$, but there are cases where it is important to *count the occurrences* of $\mathbf{x}$ in $\mathbf{S}$. A bag represents this by allowing repeated occurrences of the same element, or, in other words, by attaching a positive integer count to each element of a set. So, for example, $\{\mathbf{x}, \mathbf{x}, \mathbf{y}, \mathbf{z}\} - \{\mathbf{x}, \mathbf{y}\} = \{\mathbf{x}, \mathbf{z}\}$.

A PN is a directed graph whose nodes are partitioned into two sets, *places* and *transitions*. Arcs can only connect a place to a transition (*input arcs*), or a transition to a place (*output arcs*). A *multiplicity* (positive integer) may be attached to each arc, which is then called a *multiple arc*. Intuitively, a multiple arc with multiplicity $\mathbf{k}$ can be thought of as $\mathbf{k}$ arcs having the same source and destination.

The *input (output) bag* for a transition is the bag constituted by the input (output) arcs, considered with their multiplicity. Each place may contain any number of *tokens*. All the tokens are indistinguishable. A *marking* is a bag representing the configuration of tokens in the places of the PN. It can be thought as the state of the PN.

Regarding the evolution of the PN, the following terms are fundamental. A transition is *enabled* if its input bag is a subbag of the (current) marking. When a transition is enabled, it can *fire*, leading the PN into a different marking, obtained by subtracting its input bag from and adding its output bag to the current marking. A *firing sequence* is a sequence of transition firings. A marking is *reachable* if it is obtained by a firing sequence starting in the *initial marking*. The *reachability set (graph)* is the set (graph) of all the reachable markings (connected by arcs labeled with the transition firings). A set $\mathbf{S}$ of transitions enabled in a marking $\mathbf{m}$ is a *conflicting transition set* if the contemporary firing of all the transitions of $\mathbf{S}$ is impossible in $\mathbf{m}$, or, in other words, if the sum of the input bags of the transitions in $\mathbf{S}$ is not a subbag of $\mathbf{m}$.

The SPN model is obtained from the PN model by associating a *probability distribution*

function to the firing time of each transition. Additional constructs are often present as well. In the GSPN model [2], only two distribution types are allowed: exponential and deterministic with value 0. Transitions with an associated exponential distribution are said to be *timed*; transitions with a constant 0 distribution are said to be *immediate*. In the ESPN model [10], the transitions are classified in a similar way, but an arbitrary distribution can be associated to each timed transition. If two or more conflicting transitions should fire at the same moment (this event has a 0 probability if the distribution is continuous), a *probability function* must specify the probability that a subset of transitions will actually fire. When the implementation is completed, our model will be the ESPN, but for the moment it is limited to the GSPN, so we prefer to use the general term SPN.

The parameter(s) of an exponential or general distribution are said to be *marking dependent* if they can be different in each marking. This is allowed by our definition; we will show how to describe this marking dependency.

Additional constructs are used to selectively disable a transition in a marking which would otherwise enable it. A *priority* is associated with each transition. If **S** is the set of transitions enabled in a marking and if the transition with the highest priority among them is **k**, then any transition in **S** with priority lower than that of transition **k** will be disabled. To avoid theoretical difficulties, timed and immediate transitions cannot have the same priority. Another way to disable a transition is the *inhibitor arc*. An inhibitor arc from place **p** to transition **t** with multiplicity **m** will disable **t** in any marking where **p** contains at least **m** tokens. If these two constructs are not sufficient to describe a particular mechanism, the marking dependent *enabling function* associated with each transition can be used: if this function evaluates to 0 in a marking, then the transition is disabled. If we ignore timing, we can imagine an ordinary PN as a SPN where all transitions have the same priority, where no inhibitor arcs are present, and where the enabling functions are identically equal 1.

A marking is *tangible* if it enables no immediate transition; it is called a *vanishing* marking otherwise. A marking which does not enable any transition is *absorbing*, hence it is tangible by definition. A (maximal) set of vanishing markings that are mutually reachable by immediate transition firings is called a *loop* (of vanishing markings). A loop is said to be *absorbing* if no marking in it reaches a marking outside the loop; otherwise the loop is *transient*. An absorbing loop is considered an error. Transient loops are not a problem, their interpretation is clear, and they are easily and correctly managed by the package, but, if you know that your SRN should not contain a transient loop, you should look for them, since they could be the manifestation of a modeling error. The reachability graph contains an arc for each different transition enabled in each marking: in particular, *self transitions* (with equal input bags and output bags) are allowed by the definition of the model, so arcs with coinciding source and destination may be present in the reachability graph. These arcs are ignored during the solution steps.

Another feature of our model is the *marking dependent arc multiplicity*. Arcs can have a

multiplicity which is not constant, but rather it is a function of the marking. This possibility was defined because it may allow substantial reductions in the size of the reachability graph; it also may allow to model, in a compact way, behaviors that would otherwise require complex subnets. A typical example is the case where *all* the tokens from place **p** must be moved to place **q** when transition **t** fires. An input arc from **p** to **t** and an output arc from **t** to **q**, both with marking dependent multiplicity equal to the number of tokens in place **p** are enough to model this behavior. Without this construct, the reachability graph would contain all the intermediate arcs and markings corresponding to the movement of tokens, *one by one*. Perhaps even more importantly, if **t** is timed, the stochastic behavior will not be the same, unless the SRN explicitly models this "flushing" of tokens with an additional immediate transition and possibly some control places. Some words of caution must be said on this construct. First, it should be used only when really needed, because it may make the SRN harder to understand and it and it requires more computation that a standard or multiple arc. Then, the input and output bags for a transition in a marking are computed by evaluating the marking dependent functions for the arc multiplicities, if any, *before firing the transition* (this is why the output arc from **t** to **q** will put the correct amount of tokens in **q**). This might give rise to unintuitive or unforeseen behaviors; for example, in the flushing of tokens just described, **t** is enabled in any marking, even when **p** is empty, unless (1) other input arcs are defined for **t**, (2) an enabling function is used to explicitly disable **t** when **p** is empty and possibly in other cases as well, or (3) the marking dependent arc multiplicity function for the arc from **p** to **t** returns a positive value when **p** is empty (this is the most efficient solution if the goal is to enable **t** only when there are tokens to be flushed in **p**).

3 Structure of the package

The package is composed of several C files. The SRN to be studied must be described in a CSPL (C-based Stochastic Petri Net Language) file, which is a C file specifying the structure of the SRN and the desired outputs, by means of predefined functions. The CSPL file is compiled, linked to the other files constituting the package (usually kept compiled), and run, by typing

```
make -f /PATH_TO_SPNP/spnp/obj/Makerun SRN=filename
```

where `filename` is your CSPL file, without the C extension, and `PATH_TO_SPNP` represents the UNIX path to the package on your installation.

The intermediate files generated by the package and the final results will be in the directory where **filename.c** is (and where you issued the command). Files have different extensions, according to the kind of information they carry. If your CSPL file is named **test.c**, then the following files can be generated:

- **test.o** obtained when compiling **test.c**.
- **test.rg** containing the reachability graph information: composition of each marking, description of the transition firings between them, etc. (generated only if **IOP_PR_RG** has value **VAL_YES** or if **IOP_MC** has value **VAL_CTMC**).
- **test.mc** containing the (numerical) CTMC/DTMC corresponding to the SRN (generated only if **IOP_PR_MC** has value **VAL_YES**).
- **test.prb** containing the (numerical) results of the analysis of the underlying CTMC: the probabilities for each tangible marking. Currently, only the steady-state probabilities for irreducible CTMC can be obtained (generated only if **IOP_PR_PROB** has value **VAL_YES**).
- **test.out** containing the requested output (according to what is specified in **test.c** using the provided functions).
- **test.spn** executable file obtained by linking the package object files together with **test.o**.
- **test.log** contains all the output messages produced by the package during model solution.

It is possible to check the CSPL file for certain errors, by typing

```
make -f /PATH_TO_SPNP/spnp/obj/Makerun lint SPN=filename
```

If inconsistencies exist between the definition of the predefined functions and their usage, they will be discovered.
Note: To save typing, you can define the aliases

```
alias spnp "make -f /PATH_TO_SPNP/spnp/obj/Makerun SPN=\!^"
alias spnpcheck "make -f /PATH_TO_SPNP/spnp/obj/Makerun lint SPN=\!^"
```

(assuming that "!" is your history character) and type, regardless of your current directory,

```
spnp filename
spnpcheck filename
```

4 CSPL

You must describe your SRN in CSPL. The syntax and the semantics of CSPL are based on the C language, moreover a (correct) CSPL file is a (correct) C file too. What distinguishes CSPL from C is a set of predefined functions available for the definition of SRN entities. If you are not familiar with C you might not be able to exploit all the features of the language, but you should be able to describe most of your SRN without difficulty, especially using the files provided as examples (named **example1.c**, **example2.c**, etc.).

Note that any legal C construct can be used, as needed. You can define your own variables and functions and use them anywhere in the CSPL file. In particular, all the C library functions, such as **fprintf**, **fscanf**, **log**, **exp**, etc., perform as expected. The following types are predefined, for clarity purposes only: **enabling_type**, **probability_type**, **rate_type**, and **reward_type**; the first type is an alias for the C type **int** (integer number), meant to assume values **VAL_YES** and **VAL_NO** only; the others are aliases for the C type **double** (double-precision floating point number).

Several constants and functions are also predefined, they will be described as needed.

A CSPL file must specify the following functions: **parameters**, **net**, **assert**, **ac_init**, **ac_reach**, and **ac_final**.

In **parameters**, function calls to **iopt**, **fopt**, and **input** can be present. Their syntax and meaning are explained in the following.

void iopt(*option*,*value*)
int *option*,*value*;

void fopt(*option*,*value*)
int *option*;
double *value*;

iopt and **fopt** set option *option* to have value *value*. The available options are listed in a later section of this manual. You should always be sure to use the right function, **iopt** for an integer option, **fopt** for a double-precision floating point option, and to pass to it a value of the right type (respectively **int** or **double**). For example:

```
parameters() {
   ......
   iopt(IOP_PR_MARK_ORDER,VAL_LEXICAL);
   ......
}
```

will cause the markings to be printed in lexical order (see later), instead of the default

canonical order, the order in which they are found.[1]

double input(*msg*)
char **msg*;

Use it to input a parameter at run time, when **parameter** is called. A message of the form **Please type** "*msg*" is displayed on the screen (more precisely, on the **stderr** stream), and the program waits for you to type a value (more precisely, it attempts to read a double-precision floating point constant from the **stdin** stream). The assigned value is printed in the ".out" file, together with *msg*, this is useful to recall the set of values input to a particular CSPL file to generate the current output. The returned value is a **double**, so an explicit type conversion to **int** or **double** may be needed.

In **net**, function calls to **place**, **init**, **trans**, **priority**, **enabling**, **rateval**, **ratedep**, **ratefun**, **probval**, **probdep**, **probfun**, **srateval**, **sratedep**, **sratefun**, **sprobval**, **sprobdep**, **sprobfun**, **iarc**, **oarc**, **harc**, **miarc**, **moarc**, **mharc**, **viarc**, **voarc**, and **vharc** can be present. Their syntax and meaning are explained in the following.

void place(*name*)
char **name*;

void trans(*name*)
char **name*;

Define a place (a transition) with name *name*. A name is legal if: (1) its length is between 1 and **MAXNAMELENGTH**, as defined in the file **const.h**[2], usually equal 20; (2) it is composed of the characters {**0..9,a..z,A..Z,_**} only; (3) the first character is in {**a..z,A..Z**}. All names must be distinct, that is, it is an error to have two places, two transitions, or a place and a transition with the same name. Note that the keywords automatically written by the package in the intermediate files will always have an underscore as first character, to avoid possible conflicts with legal names (and to allow an easy parsing of the intermediate files).

void init(*name*,*n*)
char **name*;
int *n*;

Defines the initial number of tokens in place *name* to be *n*. By default, places are otherwise initially empty (0 tokens).

[1] **IOP_PR_MARK_ORDER** and **VAL_LEXICAL** are predefined integer constants known to the package.

[2] The definitions in the files **const.h** and **type.h** are always provided with the distribution of the package, but they should be changed only if the source code is available, since they are included in the other source files, hence used at compile-time.

void priority(*name*,*prio*)
char **name*;
int *prio*;

void enabling(*name*,*efunc*)
char **name*;
enabling_type (**efunc*)();

Defines the priority (the enabling function) for transition *name* to be *prio* (*efunc*). By default transitions have the lowest priority (0) and have no enabling function (or better, their enabling function is set to the constant 1).

void rateval(*name*,*val*)
char **name*;
rate_type *val*;

void ratedep(*name*,*val*,*pl*)
char **name*;
rate_type *val*;
char **pl*;

void ratefun(*name*,*func*)
char **name*;
rate_type (**func*)();

void probval(*name*,*val*)
char **name*;
probability_type *val*;

void probdep(*name*,*val*,*pl*)
char **name*;
probability_type *val*;
char **pl*;

void probfun(*name*,*func*)
char **name*;
probability_type (**func*)();

void srateval(*name*,*val*,*dval*)
char **name*;
rate_type *val*,*dval*;

void sratedep(*name*,*val*,*dval*,*pl*)
char **name*;
rate_type *val*,*dval*;

char *pl;

void sratefun($name$,$func$,$dfunc$)
char *$name$;
rate_type (*$func$)(),(*$dfunc$)();

void sprobval($name$,val,$dval$)
char *$name$;
probability_type val,$dval$;

void sprobdep($name$,val,$dval$,pl)
char *$name$;
probability_type val,$dval$;
char *pl;

void sprobfun($name$,$func$,$dfunc$)
char *$name$;
probability_type (*$func$)(),(*$dfunc$)();

Define the rate (probability) of transition $name$ as a constant value val of type **rate_type** (**probability_type**) – remember that they are both aliases for **double**, or double-precision floating point number –, or as a constant value val times the number of tokens in place pl – an infinite server behavior –, or as a general marking dependent function $func$ returning a **rate_type** (**probability_type**). This also implicitly defines the type of transition, timed or immediate, since rates can be defined only on timed transitions, and probabilities only on immediate transitions. When *sensitivity analysis* is needed, both the rate (probability) and its derivative need to be specified. The latter six functions (all starting with s) specify the rate (probability) and its derivative. Exactly one of the above twelve functions must be used for each defined transition (no default type, rate, or probability exists).
Note: You cannot disable a transition by defining a rate or a probability that evaluates to 0 in the marking; the package exits with an error message if a non-positive rate or probability is found for a transition which would be otherwise enabled. You must explicitly disable a transition (for example using the **enabling** function) in this situation.

void iarc(t_name,p_name)
char *t_name,*p_name;

void oarc(t_name,p_name)
char *t_name,*p_name;

void harc(t_name,p_name)
char *t_name,*p_name;

void miarc(t_name,p_name,$mult$)
char *t_name,*p_name;

int *mult*;

void moarc(*t_name*,*p_name*,*mult*)
char **t_name*,**p_name*;
int *mult*;

void mharc(*t_name*,*p_name*,*mult*)
char **t_name*,**p_name*;
int *mult*;

void viarc(*t_name*,*p_name*,*func*)
char **t_name*,**p_name*;
int (**func*)();

void voarc(*t_name*,*p_name*,*func*)
char **t_name*,**p_name*;
int (**func*)();

void vharc(*t_name*,*p_name*,*func*)
char **t_name*,**p_name*;
int (**func*)();

Define respectively an input, output, or inhibitor arc from transition *t_name* to place *p_name*, with multiplicity 1, constant *mult* (a positive integer), or given by the marking dependent function *func* (returning an integer). Note that we chose to define **vharc** for completeness, but it is usually more efficient to use an enabling function instead.

The functions passed as actual parameters to **enabling**, **ratefun**, **probfun**, **viarc**, **voarc**, or **vharc** must be defined in the CSPL file before being used. They express marking dependency using the following predefined functions:

int mark(*p_name*)
char **p_name*;

returning the number of tokens in place *p_name*.

int enabled(*t_name*)
char **t_name*;

returning 1 if transition *t_name* is enabled, 0 otherwise.

For example, the CSPL file of figure 1 defines a SRN with two places, "left_place" and "right_place", and two transitions with exponentially distributed firing times, "from_left_to_right" and "from_right_to_left". The rate of the first transition is determined by the function *myval* to be 7.3 times the number of tokens in place "left_place" while the rate of the second transition is the constant 1.0. Since the first transition has an infinite-server behavior, we would have achieved the same effect with

```
ratefun_type    myval() { return(mark("left_place") * 7.3); }
parameters() {}
net() {
   place("left_place");                  place("right_place");
   init("left_place",4);
   trans("from_left_to_right");          trans("from_right_to_left");
   ratefun("from_left_to_right",myval);
   rateval("from_right_to_left",1.0);
   iarc("from_left_to_right","left_place");
   iarc("from_right_to_left","right_place");
   oarc("from_left_to_right","right_place");
   oarc("from_right_to_left","left_place");
}
assert() { return(RES_NOERR); }
ac_init() {}
ac_reach() {}
ac_final() { pr_std_average(); }
```

Figure 1: A simple CSPL file

```
ratedep("from_left_to_right",7.3,"left_place");
```

which would have been more efficient, and perhaps more clear.

WARNING: an error arises if "1" is used instead of "1.0", in the specification of the constant rate for the second transition, because the procedure call conventions of C do not provide automatic type conversion in this case. This kind of errors is not caught by the C compiler ("spnp" command), but it is caught by "lint" ("spnpcheck" command).

The language is powerful: using the **for** construct of C, for example, it is possible to define an array of places, transitions, or arcs in a compact way. The portion of CSPL file in figure 2 allows the run-time definition of the number of stages in a subnet corresponding to an Erlang distribution. The size of the arrays of "ph" places and "th" transitions, is determined respectively as $max + 1$ and max at run-time using the predefined **input** function. You will see a message on the terminal

```
Please type 'number of phases' >
```

and max will be set to the number you type, let's say 6. Then seven places with names "ph0", ..., "ph6" will be defined, with no tokens in them initially. Six transitions will be defined, with names "th1", ..., "th6". Finally, a sequence of arcs ("ph0","th1"), ("th1","ph1"), ..., ("th6","ph6") will be defined as well.

The CSPL file must contain other functions: **assert**, **ac_init**, **ac_reach**, and **ac_final**. **assert** is called during the reachability graph construction, to check the validity of each newly found marking. It must return **RES_ERROR** if the marking is illegal, **RES_NOERR** if the marking is (thought to be) legal (**RES_ERROR** and **RES_NOERR** are predefined values). The check on the legality of a marking is performed using the same functions used to achieve marking dependency, namely **mark** and **enabled**. The check is by its own nature incomplete, since it is not usually feasible to specify all the conditions that must hold (or not hold) in a marking, but the more accurate the set of conditions is, the more confidence you should have in the correspondence of the reachability graph with the real system. For example the **assert** definition in figure 1 will not perform any check, while

```
assert() {
   if (mark("p2") + mark("p3") != 4 || enabled("t11") && enabled("t7"))
      return(RES_ERROR);
   return(RES_NOERR);
}
```

will stop the execution in a marking where the sum of the number of tokens in places **p2** and **p3** is not 4, or where **t11** and **t7** are both enabled. If the execution is stopped, the program outputs information useful to debug the CSPL file, then it exits. If the illegal marking is caused by an unforeseen sequence of transition firings, finding that sequence using the output

```
int   max;
parameters() { max = input("number of phases"); }
net() {
   int   i;
   char  auxplace[20],auxtrans[20];
   ...
   sprintf(auxplace,"ph0");
   place(auxplace);
   for (i = 1; i <= max; i++) {
      sprintf(auxtrans,"th%d",i);
      trans(auxtrans);
      rateval(auxtrans,2.0);
      iarc(auxtrans,auxplace);
      sprintf(auxplace,"ph%d",i);
      place(auxplace);
      oarc(auxtrans,auxplace);
   }
}
```

Figure 2: An Erlang subnet

information is usually a fast process even in large reachability graphs (tens of thousands of markings).

This type of check is limited, since it helps detect the presence of illegal markings, or illegal firing sequences, but it cannot detect the absence of legal markings, or legal firing sequences (which relates to the reachability set, or graph, as a whole, and cannot be checked while the reachability graph is being built). It is important to be able to perform checks of illegality as soon as possible, typically to debug a net which is supposed to be bounded, but it is turns out not be. The examination of the whole (infinite) reachability graph is out of the question since the program will terminate printing a message for insufficient memory. Peterson [25] suggests to use the symbol ω as a substitute for the number of tokens in unbounded places, but this can be applied to standard PN only, not to PN with inhibitor arcs, where boundedness is semi-decidable.

The other functions are called at different times during the analysis, to allow user-requested actions and outputs.

ac_init is called before starting the reachability graph construction. The call

```
pr_net_info();
```

can be used in it, to output data about the SRN on the ".out" file. This is especially useful when the number of places or transitions is defined at run time (otherwise it is merely a summary of the CSPL file).

ac_reach is called after the reachability graph construction has completed. The call

```
pr_rg_info();
```

can be used in it, to output data about the reachability graph on the ".out" file (this does not affect the generation of the ".rg" file).

When either steady-state analysis or steady-state sensitivity analysis is requested, **ac_final** is called after the solution of the CTMC has completed, to allow user-requested outputs. The call

```
pr_mc_info();
```

can be used in it, to output data about the CTMC and its solution. In addition, the desired output measures must be requested in this function. Predefined functions exist to print information about each place and transition, or to allow the specification of user-defined measures: **pr_std_average** and **pr_expected** (also **pr_std_average_der** and **pr_der_expected** if steady-state sensitivity is performed).

```
pr_std_average();
```

outputs, for each place, the probability that it is not empty and its average number of tokens; for each timed transition, the probability that it is enabled and its average throughput. The average throughput $E[T_a]$ for transition a is defined as

$$E[T_a] = \sum_{i \in R(a)} p(i) * \rho(a, i)$$

where $R(a)$ is the subset of reachable markings that enable transition a, $p(i)$ is the probability of marking i, and $\rho(a, i)$ is the rate of transition a in marking i.

```
pr_std_average_der();
```

prints the derivatives of all the above standard measures. The function **pr_expected** requires the specification of a string (which is written on the ".out" file) and of a marking dependent reward function returning a double-precision floating point number:

void pr_expected(*string*,*function*)
char **string*;
reward_type (**function*)();

The derivative of the expected value of the function can be computed using:

void pr_der_expected(*string*,*function*,*dfunction*)
char **string*;
reward_type (**function*)(),(**dfunction*)();

where (**dfunction*)() is the derivative of the marking dependent reward function (**function*)() with respect to the parameter θ. Both the expected value and its derivative can be computed by using:

void pr_sens_expected(*string*,*function*,*dfunction*)
char **string*;
reward_type (**function*)(),(**dfunction*)();

For example,

```
pr_expected("utilization",util);
```

will print on the ".out" file

```
EXPECTED: utilization = 3.2
```

if the expected value of the reward function *util* is 3.2. The reward function *util*, returning a **reward_type**, or double-precision floating point number, must have been defined prior to its usage in **pr_expected**, using the functions **mark** and **enabled** to express marking dependency. In addition, the function

rate_type rate(*t_name*)
char **t_name*;

can be used, to express the marking dependent rate of transition *t_name* (defined as 0 when the transition is not enabled in the marking). For example,

```
reward_type util() {
        return(mark("p1") * mark("p7") + mark("p3") * 1.2)
}
```

would define the utilization as the weighted (by the probability of each marking) average of the product of the number of tokens in place $p1$ and $p7$ plus the number of tokens in place $p3$ times 1.2. The function **expected** is also provided:

reward_type expected(*function*)
reward_type (**function*)();

The returned value can be used in more complex expression (using **pr_expected** would print the value, but the value itself would not be made available in the function **ac_final**). Similarly the function **der_expected** returns the derivative of the expected value:

reward_type der_expected(*function*,*dfunction*)
reward_type (**function*)(),(**dfunction*)();

NOTE: Apparently similar operations have different stochastic interpretations, hence different results, if performed at the event or at the expected value level. Continuing the previous example,

```
reward_type ep1() { return(mark("p1")); }
reward_type ep3() { return(mark("p3")); }
reward_type ep7() { return(mark("p7")); }
.......
ac_final() {
     x = expected(ep1) * expected(ep7) + expected(ep3) * 1.2;
     printf("%f",x);
}
```

will produce a different result from the one computed using `util`, because of the dependence existing (in general) between the number of tokens in `p1` and `p3`.

When transient analysis or transient sensitivity analysis is required, **ac_final** is called before the solution of the CTMC. For performing transient analysis and transient sensitivity analysis, a time point needs to be specified. This can be done through the function **time_value** in **ac_final**, defined as:

void time_value(t)
double t;

Whenever this function call is encountered, transient (transient sensitivity) analysis is performed on the CTMC. All the user-requested outputs following this function call are computed for the time t specified, until a new call to **time_value** is encountered.

Besides the expected values of the functions defined earlier, transient analysis also allows the computation of the expected accumulated values over the interval $[0, t)$ where t is the time point of interest. The corresponding functions are **pr_std_cum_average**, **pr_std_cum_average_der** for computing the expected accumulated values and their derivatives for standard measures, **pr_cum_expected**, **pr_der_cum_expected** and **pr_sens_cum_expected** for computing the expected accumulated value and its derivative for user-defined functions. We can also compute the time-averaged expected values of functions using **pr_time_avg_expected** as:

void pr_time_avg_expected(*function*)
reward_type (*$function$)();

As an example, consider the following:

```
reward type  avail() { ... }

ac_final() {
   double time_point;

   time_value(15.05);
   pr_expected("Inst. Availability",avail);
   pr_cum_expected("Total jobs lost", jobslost);

   for (time_point = 10.0;time_point <= 100.0;time_point += 10.0){
      time_value(time_point);
      pr_time_avg_expected("Interval Availability",avail);
   }
   pr_mtta("Mean time to failure");
}
```

Here, the *instantaneous availability* and *total jobs lost* is computed with time $t = 15.05$. The for loop computes the *interval availability* in the interval $[0, t)$, with t varying from 10 to 100 in increments of 10. The function **pr_mtta** computes the mean time to absorption for the SRN. This function should be used only when the underlying CTMC has absorbing states. The syntax for **pr_mtta** is:

void pr_mtta(*string*)
char *****$string$;

The function **mtta**() returns the mean time to absorption.

Another function **pr_cum_abs**, which is similar to **pr_mtta**, allows the user to compute the expected accumulated reward until absorption for a CTMC with absorbing states. To use this function, the corresponding reward rate should be specified. The syntax for this function is:

void pr_cum_abs(*string*,*function*)
char *****$string$;
reward_type (*****$function$)();

The function **cum_abs**(*function*) returns the expected accumulated reward until absorption.

Transient analysis of the underlying CTMC is dependent on the initial state probability vector. When the initial marking is tangible, the state corresponding to this marking has the initial probability of 1 and all the other states have an initial probability of 0. If the initial marking is vanishing, the initial probability vector over the states of the CTMC is automatically computed by the program. The user is also allowed to define the initial probability vector over the markings of the SRN using the function:

void set_prob_init(*function*)
reward_type (*****$function$)();

At present, user-defined initial probability vector is allowed only over the set of tangible markings.

Sometimes the user may desire to print values of functions that cannot be expressed as a simple reward definition, but as a function of the expected values of several reward functions. To facilitate this, SPNP provides a special function called **pr_value**. The corresponding syntax is:

void pr_value(*string*,*expr*)
char *****$string$;
double $expr$;

Here, *expr* could be any expression which evaluates to a floating point number. For example, if we wish to compute the ratio of two expected values `expected(qlength)` and `expected(tput)` and print the result in the output file, we can specify the following:

```
pr_value("Expected Response Time",expected(qlength)/expected(tput));
```

This would print the following in the output file:

```
VALUE: Expected Response Time = 15.7
```

Another general function **pr_message** provided in SPNP allows the user to print an arbitrary message in the .out file. The syntax for this function is:

void pr_message(*string*)
char **string*;

5 Specialized output functions

SPNP was initially aimed at the steady-state solution of SRNs whose underlying CTMC is ergodic. There are a number of measures which could be considered "unusual", but closely related to steady-state. In particular, they do not require the implementation of a new solver; they can be computed either from the steady-state probabilities, or by solving a slightly different (non-homogeneous) linear system.

These measures were defined and implemented to perform decomposition–iteration techniques allowing the approximate solution of SRNs whose state-space is too large to be studied directly [7, 9]. They are **accumulated**, **pr_accumulated**, **hold_cond**, **pr_hold_cond**, **set_prob0**, and **scale_prob0**.

reward_type accumulated(*function*)
reward_type (**function*)();

void pr_accumulated(*string*,*function*)
char **string*;
reward_type (**function*)();

respectively return and print the expected value of the "accumulated reward up to absorption", according to the given initial state probability distribution, reward rate assignment, and absorbing marking definition. The initial state probability distribution must be specified by calling **set_prob0** or **scale_prob0** first. The reward rate assignment is specified, as usual, by *function*. The specification of the absorbing markings requires some attention. Since the SRNs normally managed by the package are ergodic, no absorbing markings may be present. The underlying stochastic process is then modified (for the computation of this

measure only) so that markings whose reward is null are assumed absorbing (their outgoing arcs in the Markov chain are ignored). If absorbing markings do indeed exist in the original SRN, *function* must evaluate to zero in them (otherwise the accumulated reward would be infinite). Each call to **accumulated** (or **pr_accumulated**) requires the solution of a non-homogeneous linear system having as many variables as the non-zero-reward markings, so it can be expensive.

void set_prob0(*function*)
reward_type (**function*)();

void scale_prob0(*function*)
reward_type (**function*)();

are used to set the initial state probability for the computation of the accumulated reward up to absorption. One of them must be called before each call to **accumulated** or **pr_accumulated**. Let's define ρ_i as the value returned by *function* on marking i and π_i as the steady-state probability for marking i ($\pi_i = 0$ if marking i is vanishing). A call to **set_prob0** defines the initial state probability for state i to be proportional to ρ_i:

$$\pi(0)_i = \frac{\rho_i}{\sum_j \rho_j}$$

A call to **scale_prob0** defines the initial state probability for state i to be proportional to $\rho_i \pi_i$:

$$\pi(0)_i = \frac{\rho_i \pi_i}{\sum_j \rho_j \pi_j}$$

The definition of this second function may at first seem arbitrary; it is instead both useful and intuitive. Assume that, given an ergodic SRN in steady-state, we want to know how long we need to wait before a token arrives in place `p`: the following portion of CSPL accomplishes this:

```
reward_type one()    { return(1.0); }
reward_type empty() { return( mark("p") > 0 ? 0.0 : 1.0 ); }
reward_type full()  { return( mark("p") > 0 ? 1.0 : 0.0 ); }
.......
ac_final() {
     scale_prob0(empty);
     pr_accumulated("Wait time",full);
     scale_prob0(one);
     pr_accumulated("Wait time",full);
}
```

The first output gives the waiting time *given that no token is in* `p`, while the second output gives the unconditional waiting time (that is, including the possibility that a zero waiting time is required, when a token is already in `p`).

Functions **hold_cond** and **pr_hold_cond** respectively compute and print the expected time a condition holds true or false in steady-state:

void hold_cond(*cond*,*times*)
boolean_type (**cond*)();
double *times*[2]

void pr_hold_cond(*string*,*cond*)
char **string*;
boolean_type (**cond*)();

cond must be a marking-dependent function returning **VAL_YES** if the condition holds in the marking, **VAL_NO** otherwise. On return, *times*[**VAL_YES**] and *times*[**VAL_NO**] respectively contain the expected length of time the condition holds or does not hold in steady-state. The idea behind this measure is to be able to condense a large Markov chain into a two-state process. Normally the process is not a Markov chain, but the two-state Markov chain whose transition rates are $1/times$[**VAL_YES**] and $1/times$[**VAL_NO**] can at least be considered an approximate representation of it. The description of how this measure is computed gives additional insight. Define S_y and S_n to be the sets of markings where the condition is true and false respectively and define T to be the set of tangible markings. If **IOP_MC** has value **VAL_CTMC**, *times*[**VAL_YES**] and *times*[**VAL_NO**] are computed respectively as

$$\frac{\sum_{k \in S_y \cap T} \pi_k}{\sum_{i \in S_y \cap T, j \in S_n \cap T} \pi_i \lambda_{i,j}} \quad \text{and} \quad \frac{\sum_{k \in S_n \cap T} \pi_k}{\sum_{i \in S_n \cap T, j \in S_y \cap T} \pi_i \lambda_{i,j}}$$

where $\lambda_{i,j}$ is the transition rate from marking i to marking j. If **IOP_MC** has value **VAL_DTMC**, *times*[**VAL_YES**] and *times*[**VAL_NO**] are computed respectively as

$$\frac{\left(\sum_{k \in S_y \cap T} \pi_k\right)\left(\sum_{k \in T} p_k h_k\right)}{\sum_{i \in S_y, j \in S_n} p_i \alpha_{i,j}} \quad \text{and} \quad \frac{\left(\sum_{k \in S_n \cap T} \pi_k\right)\left(\sum_{k \in T} p_k h_k\right)}{\sum_{i \in S_n, j \in S_y} p_i \alpha_{i,j}}$$

where $\alpha_{i,j}$ is the transition probability from marking i to marking j, p_j is the steady-state probability of marking j for the DTMC, and h_k is the holding time in state k for the CTMC.

It is interesting to note that DTMC and CTMC solution may give different results for this measure. The reason is intrinsic to the different approach, is not due to an error nor to numerical roundoff or truncation. If the condition holds in tangible markings m_1 and m_3 and it does not hold in vanishing marking m_2, a path (m_1, m_2, m_3) in the reachability graph is treated differently by the two approaches. The DTMC solution considers the holding time as terminated and restarted every time the path is traversed, while the CTMC solution does not know that the condition stops holding, even if for a null amount of time, when a transition from m_1 to m_3 occurs (this information is discarded together with m_2 when eliminating the vanishing markings). The holding time computed by the DTMC solution can be shorter than the one computed by the CTMC solution. In practically all interesting applications, the condition holds or does not hold for a positive amount of time with probability one, so no inconsistencies can arise.

6 Advanced CSPL features

Some features implemented in CSPL are particularly powerful and, unfortunately, complex. Marking dependency, for example, can be difficult to master and exploit at the beginning, especially when applied to input and output arcs. Another feature we have already at least partially discussed is the ability to define at run-time certain characteristics of the SRN, even the presence of a place or the priority of a transition, using the **input** function. This can be stretched to the point where a single CSPL file can actually represent a class of SRNs, not a single one (maybe CSPL should be called a meta-SRN language), as in the Erlang subnet example.

What was actually done in the Erlang subnet was the generation of an array of places and transitions (with the appropriate names), but to do this, an array of names had to be generated using the system function **sprintf**. Furthermore, the package had no internal knowledge that the places (the transition) where in some sense logically connected.

This considerations prompted us to implement functions that explicitly manage arrays of places, transitions, and even arcs.

At the moment, the approach is not very elegant or efficient (although it is more elegant and efficient than the implicit approach used in the Erlang subnet example). Our choice of the C language and the desire of a fast prototype implementation imposed some shortcomings in the elegance of the design (the need to be constantly reminded of the dimensionality of each array is particularly annoying, as we will see). A higher-level language, such as LISP, would allow a more elegant specification language, although it would probably be slower at run time.

The general approach is the following. Every place and transition is considered as a two-dimensional array. An ordinary place, for example, is considered a 1×1 array; a unidimensional array of transitions, for example, is considered a $n \times 1$ array. Every predefined

function is implemented as working on two-dimensional arrays of places and/or transition. For example,

```
place("p");
```

is translated (see file "user.h") into

```
Cplace("p",1,1);
```

Most functions described in the previous section have a corresponding 0- 1- and 2-dimensional form, obtained by adding a "_0", "_1", or "_2" to the basic function. For example, the following are all legal and semantically equivalent:

```
place("p");
place_0("p");
place_1("p",1);
place_2("p",1,1);
```

To define an array of 4×5 places, with names "a.0.0" through "a.3.4" use

```
place_2("a",4,5);
```

Note that, to indicate a single place in the array, the name and the indexes are used separately (not merged using **sprintf** as in the Erlang subnet example):

```
init_2("a",1,1,3);
```

will set the initial number of tokens in place "a.1.1" to 3.

Sometimes a function may need to be applied to all the elements of an array. A possible solution is to use the **for** construct of C, as in

```
trans_1("t",30);
for (i = 0; i < 30; i++)
    priority_1("t",i,7);
```

but a simpler way is

```
trans_1("t",30);
priority_1("t",ALL,7);
```

which achieves the same results and it is both more clear and more efficient.

The specification of input, output, or inhibitor arcs is particularly complex. First of all, there are nine possible cases, according to the dimensionality of both the transition and the place arrays. So, for example,

```
miarc_2_1("t",4,5,"p",2,3);
```

specifies an input arc with multiplicity 3 from "p.2" to "t.4.5". Another difficulty is the inadequacy of the **ALL** keyword, shown in the following example:

```
trans_2("t",6,7);
place_1("p",6);
place_0("q");
place_2("m",6,7);
iarc_2_1("t",3,ALL,"p",3);
iarc_2_0("t",ALL,ALL,"q");
iarc_2_2("t",ALL,ALL,"m",ALL,ALL);
```

The first "iarc" statement defines seven input arcs, from "p.3" to "t.3.0" through "t.3.6". The second "iarc" statement defines 42 input arcs, from "q" to "t.0.0" through "t.5.6". The third "iarc" statement defines 1764 input arcs, from each of "m.0.0" through "m.5.6" to each of "t.0.0" through "t.5.6", while probably the intended behavior was to define 42 arcs only, from "m.i.j" to "t.i.j", for all the legal values of i and j, which should have been specified as

```
for (i = 0; i < 6; i++)
    for (j = 0; j < 7; j++)
        iarc_2_2("t",i,j,"m",i,j);
```

Clearly the range of possible connection patterns between places and transitions is so large that the only general approach is to require explicit declaration, arc by arc, often using the **for** loop of C. The **ALL** keyword can be used in the cases where it represents the intended behavior.

Two issues connected with the arrays arise in the definition of the marking dependent functions. The first is the need to specify the behavior of a whole array of transitions (e.g. rate) with a single function, since the purpose of allowing arrays would be at least partially defeated if a different function had to be defined for each transition in the array. Yet, each transition in the array may have a slightly different behavior. The solution is to pass to the function itself the indexes of the transition in the array. For example:

```
trans_2("t",3,4);
ratefun_2("t",ALL,ALL,fun);
```

declares that the 12 transitions in the array have a marking dependent rate given by `fun`, but, more precisely, the rate of "t.0.0" is given by `fun(0,0)`, the rate of "t.0.1" is given by `fun(0,1)`, and so on. The definition of `fun` could be:

```
rate_type fun(i,j) {
    int a,b;
    if (i == 0) {
        return(mark_1("p",j) * 3.2);
    } else {
        a = mark_1("a",j);
        b = mark_1("b",j);
        return((a > b) ? a * 2.6 : b * 1.2);
    }
}
```

which of course assumes that 1-dimensional arrays of places "p", "a", and "b" exists and have dimension at least 4.

The other issue is connected to the use of the functions **mark** and **enabled**. It is easy to imagine situations where it would be desirable to define these function as having a meaning when applied to the whole array, rather than to a single element, similarly to what the ALL keyword allows. Three keywords have been defined, to be used in this contest: **SUM**, **MAX**, and **MIN**. For example,

```
mark_2("p",SUM,SUM);
```

evaluates to the total number of tokens in the array "p", while

```
mark_2("p",MIN,MAX);
```

evaluates to the minimum over i of the maximum over j of `mark_2("p",`i`,`j`)`. When applied to **enabled**, the keywords have the same meaning, just remember that, while the application of **enabled** to a single transition can only return 0 or 1, the application of the same function to a whole array can return an arbitrary non-negative integer if the keyword **SUM** is used. For example,

```
enabled_1("t",SUM);
```

will return the count of the enabled transitions in the array "t".

Using the **for** construct of C and the **input** function to decide at run-time the size of the array, a very large class of SRNs can be represented by a single CSPL file.

7 Format of the intermediate files

This section explains how to interpret the data in the intermediate files generated during the analysis of a SRN.

7.1 ".rg" file

This file describes the reachability graph corresponding to the SRN. The format of the information is the following:

```
_nplace = <number of places>;
_ntrans = <number of transitions>;
_places =
        <pl>: <place name>;
        ................
        <pl>: <place name>;
_transitions =
        <tr>: <transition name>;
        ....................
        <tr>: <transition name>;
_ntanmark = <number of tangible non-absorbing markings>;
_nabsmark = <number of (tangible) absorbing markings>;
_nvanmark = <number of vanishing markings>;
_nvanloop = <number of transient loops>;
_nentries = <number of arcs in the reachability graph>;
_reachset =
   <mk><lbl>      <pl>:<tk>  ...  <pl>:<tk>;
   ..................................
   <mk><lbl>      <pl>:<tk>  ...  <pl>:<tk>;
_reachgraph =
   <mk>      <mk>:<tr>:<val>  ...  <mk>:<tr>:<val>;
   ..................................
   <mk>      <mk>:<tr>:<val>  ...  <mk>:<tr>:<val>;
```

where `<mk>` is the integer index of a marking (non-negative for tangible markings, negative for vanishing markings) and `<lbl>` is a code (`_T` for tangible, non-absorbing; `_A` for tangible, absorbing; `_V` for vanishing marking not in a loop; `_L` for vanishing marking in a transient loop); `<pl>` is the non-negative integer internally assigned to each place (in the same order of definition in the CSPL file); `<tr>` is the non-negative integer internally assigned to each transition (in the same order of definition in the CSPL file); `<tk>` is the (positive) number of tokens in a place; and `<val>` is the transition rate or probability in the marking. So, for example, this row in the reachability set specification

```
3_A    0:1    6:5;
```

means that marking 3, an absorbing tangible marking, has one token in place 0 and five tokens in place 6, while this row in the reachability graph specification

```
   -4    4:2:0.7    -6:5:0.3;
```

means that marking -4, a vanishing marking, goes to marking 4 by firing transition 2 with probability 0.7, and to marking -6 by firing transition 5 with probability 0.3 (of course both transition are immediate). If the option **IOP_PR_FULL_MARK** is turned on, the format for the description of the reachability set is instead

```
_reachset =
#          <place1>   <place2>  ... <placeN>
<mk><lbl>      <tk>      <tk >  ...     <tk>
 ....................................
<mk><lbl>      <tk>      <tk >  ...     <tk>
```

7.2 ".mc" file

If **IOP_MC** has value **VAL_CTMC**, this file describes the CTMC derived from your SRN; the vanishing markings are absent and only numerical rates appear. The format is:

```
_firstindex = 0;
_nstates = <number of states>;
_nentries = <number of arcs in the CTMC>
_order = <_FROMTO or _TOFROM>;
_matrix =
   <state>       <state>:<rate>  ...  <state>:<rate>;
   ...............................................
   <state>       <state>:<rate>  ...  <state>:<rate>;
[_dermatrix =
   <state>       <state>:<rate>  ...  <state>:<rate>;
   ...............................................
   <state>       <state>:<rate>  ...  <state>:<rate>;]
_method = <requested solution method>;
_precision = <requested precision>;
[_initstate =
   <state>:<prob> ... <state>:<prob>
   ......................................
   <state>:<prob> ... <state>:<prob>;]
[_iterations = <maximum number of iterations>;
_solve = _ALL;]
[_time = <time_point>;
_solve = _ALL;]
..................
```

```
[_time = <time_point>;
_solve = _ALL;]
```

All entries enclosed in square brackets ([...]) are optional. The transition rate matrix is described by rows. If **_FROMTO** is in effect,

```
7            5:0.4       8:1.2      12:100;
```

means that the transition rate from state 7 to state 5 is 0.4, to state 8 is 1.2, to state 12 is 100.0. The first index is 0, so if the number of states is 15, they will be identified as 0,2,...,14. If the order is **_TOFROM**, the transpose of the transition rate matrix will be printed. In our example, there will be rows

```
5            ...  7:0.4  ...;
8            ...  7:1.2  ...;
12           ...  7:100  ...;
```

The matrix order, the method for the solver, the precision, and the maximum number of iterations can be changed by using the appropriate options in the CSPL file.

If **IOP_MC** has value **VAL_DTMC**, this file describes the DTMC derived from your SRN, the vanishing markings are still present and probabilities are given instead of rates (the matrix is *stochastic*).

7.3 ".prb" file

This file describes the steady-state probability for each tangible marking; it corresponds to the result of the CTMC solution (even when the actual solution used a DTMC). The format is the following:

```
_firstindex = 0;
_nstates = <same value as in input>;
_method = <method actually used>;
_totaltime = _STEADYSTATE;
_steptime = _NONE;
_precision = <the reached precision>;
_iterations = <the actual number of iterations>;
_time = <_STEADYSTATE>|<time_point>;
_probabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;
[_derprobabilities =
```

```
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
[_cumprobabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
[_dercumprobabilities =
   <state>:<prob>  ...  <state>:<prob>
   ..................................
   <state>:<prob>  ...  <state>:<prob>;]
```

The method may be changed automatically, so its value in this file reflects the actual choice, possibly different from the one declared in the ".mc" file. In the current implementation of the steady-state solver provided within the package, **_SSSOR** is changed automatically into **_GASEI** when the maximum number of iterations is reached (and the iteration count is reset to 0). Uniformization is the only transient solution method available at the present.

8 Available options

The following will list the options, their legal values, and their type: particular care must be taken when specifying the value, since, for example, 10 and 10.0 are different constants, and only one of them will be correct (10 for **iopt**, 10.0 for **fopt**). Use the **spnpcheck** command to discover these errors.

- **IOP_PR_MARK_ORDER** specifies the order in which the markings are printed. With **VAL_CANONIC** order, markings are printed in the order they are found, in a breadth-first search starting from the initial marking, and in increasing order of enabled transitions indices. It is the most natural order and it is particularly helpful when debugging the SRN. With **VAL_LEXICAL** order, markings are printed in increasing order, where marking are compared as words in a vocabulary, the possible number of tokens being the alphabet, and the order of the "letters" in a "word" being given by the order of the non-empty places in the marking: for example (2_T 3:2 4:1 5:1) comes before (3_A 3:2 4:3 6:1). This order may be useful when searching for a particular marking in a large ".rg" file, although an editor with search capabilities used with the **VAL_CANONIC** order is usually adequate for the purpose. With **VAL_MATRIX** order, markings are printed in the same order as the states of the two Markov chains built internally: the DTMC corresponding to the vanishing markings, and the CTMC corresponding to the tangible markings. This corresponds to the following ordering:

type	name	values	default
int	IOP_PR_MARK_ORDER	VAL_CANONIC VAL_LEXICAL VAL_MATRIX	VAL_CANONIC
int	IOP_PR_MERG_MARK	VAL_YES VAL_NO	VAL_YES
int	IOP_PR_FULL_MARK	VAL_YES VAL_NO	VAL_NO
int	IOP_PR_RSET	VAL_YES VAL_NO VAL_TANGIBLE	VAL_NO
int	IOP_PR_RGRAPH	VAL_YES VAL_NO	VAL_NO
int	IOP_PR_MC	VAL_YES VAL_NO	VAL_NO
int	IOP_PR_MC_ORDER	VAL_FROMTO VAL_TOFROM	VAL_FROMTO
int	IOP_PR_PROB	VAL_YES VAL_NO	VAL_NO
int	IOP_MC	VAL_CTMC VAL_DTMC	VAL_CTMC
int	IOP_OK_ABSMARK	VAL_YES VAL_NO	VAL_NO
int	IOP_OK_VANLOOP	VAL_YES VAL_NO	VAL_NO
int	IOP_OK_TRANS_M0	VAL_YES VAL_NO	VAL_YES
int	IOP_METHOD	VAL_SSSOR VAL_GASEI VAL_TSUNIF	VAL_SSSOR
int	IOP_CUMULATIVE	VAL_YES VAL_NO	VAL_YES
int	IOP_SENSITIVITY	VAL_YES VAL_NO	VAL_NO
int	IOP_ITERATIONS	non-negative **int**	2000
int	IOP_DEBUG	VAL_YES VAL_NO	VAL_NO
int	IOP_USENAME	VAL_YES VAL_NO	VAL_NO
double	FOP_ABS_RET_M0	non-negative **double**	0.0
double	FOP_PRECISION	non-negative **double**	0.000001

Table 1: Available options

vanishing, tangible non-absorbing, and tangible absorbing, each of these group ordered in canonical order.

- **IOP_PR_MERG_MARK** specifies whether the tangible and vanishing markings must be printed together, or two separate lists must be printed.
- **IOP_PR_FULL_MARK** specifies whether the markings are printed in long format (a full matrix indicating, for each marking, the number of tokens in each place, possibly zero), or short format (for each marking, a list of the number of tokens in the non-empty places). **VAL_YES** looks good only when the SRN has a small number of places.
- **IOP_PR_RSET** and **IOP_PR_RGRAPH** specify whether the reachability set and graph must be printed. **VAL_TANGIBLE** specifies that only the tangible markings must be printed; it cannot be used for **IOP_PR_RGRAPH**.
- **IOP_PR_MC** specifies whether the ".mc" file is generated or not.
- **IOP_PR_MC_ORDER** specifies whether the transition matrix (if **VAL_FROMTO**) or its transpose (if **VAL_TOFROM**) is printed in the ".mc" file.
- **IOP_PR_PROB** specifies whether the ".prb" file is generated or not.
- **IOP_MC** specifies the solution approach. Using **VAL_CTMC** will transform the SRN into a CTMC. Using **VAL_DTMC** will use an alternative solution approach, where the vanishing marking are not eliminated and a DTMC is instead solved. In this case, the first index in the ".mc" file is $-n$, if there are n vanishing markings, not 0. The package performs transient and sensitivity analysis by reducing the SRN to CTMC. Hence this option should be set to **VAL_CTMC** when these types of analysis is needed.
- **IOP_OK_ABSMARK**, **IOP_OK_VANLOOP**, and **IOP_OK_TRANS_M0** specify respectively whether absorbing markings, transient vanishing loops, and a transient initial marking are acceptable or not. If **VAL_NO** is specified, the program will stop if the condition is encountered. If **VAL_YES** is specified, the program will signal such occurrences, but it will continue the execution.
- **IOP_METHOD** allows to set the numerical solution method for the CTMC, **VAL_SSSOR** stands for Steady State SOR, **VAL_GASEI** stands for Steady State Gauss-Seidel, **VAL_TSUNIF** stands for Transient Solution using Uniformization. Note that there are cases where SOR does not converge, while Gauss-Seidel converges, and vice versa.
- **IOP_CUMULATIVE** specifies whether cumulative probabilities should be computed.

- **IOP_SENSITIVITY** specifies whether sensitivity analysis should be performed.
- **IOP_ITERATIONS** specifies the maximum number of iterations allowed for the numerical solution.
- **IOP_DEBUG** causes the output (on the "stderr" stream) of the markings as they are generated, and of the transitions enabled in them. It is extremely useful when debugging a SRN.
- **IOP_USENAME** specifies whether the names must be used to indicate the places and transitions involved when printing the reachability set and graph, instead of the index (a small integer starting at 0). Using names generates a larger ".rg" file and prevents its subsequent parsing (in the current version), but it is useful when debugging a SRN.
- **FOP_ABS_RET_M0** specifies the value of the rate from each absorbing marking back to the initial marking. If this rate is positive, these markings will not correspond to absorbing states in the CTMC. This is useful to model a situation that would otherwise require a large number of transitions to model this "restart". Of course the numerical results will depend on the value specified for this option.
- **FOP_PRECISION** specifies the minimum precision required from the numerical solution. The numerical solution will stop either if the precision is reached, or if the maximum number of iteration is reached. Both the reached precision and the actual number of iterations are always output in the ".prb" file, so you can (and should) check how well the numerical algorithm performed.

9 Planned work

- A simulation solution method will be available. In this case the reachability set is not generated (though **assert** can still be used). An executable file **test.sim** will be generated by linking **test.o** with other object files. The output of **test.sim** will be a file **test.out**, analogous to the one obtained from the analytical solution, but with confidence intervals instead of point values.
- General or even empirical distributions will be allowed in conjunction with simulation. A random deviate from the distribution might be needed for simulation, but even in the presence of general distributions, an analytical solution may sometimes be feasible, so the parameters of the distribution are needed, not a random deviate, just like in the exponential case. A stable definition will be possible only after the implementation is completed, and some experience from using the package is gained.

- The definition of a higher-level language, or especially the availability of a graphical input tool, will improve user-friendliness.
- Global checks on the absence of legal markings or firing sequences could be performed after the ".rg" file is generated. A reasonable place to define these checks is **ac_reach**, since it is called after the reachability graph construction. In **ac_reach** you should be able to specify the checks to be performed on the reachability graph, such as making sure that a certain (sub)marking is matched or covered, and even list the markings matching or covering it.
- An alternative standard format containing the distributions, instead of the averages, is under preparation.
- The ".rg" file is always generated if **IOP_MC** has value **VAL_CTMC**; an alternative data structure to avoid this write-and-read-back process is being considered (if **IOP_MC** has value **VAL_DTMC**, the ".rg" file need not be generated). An additional problem is that, if **IOP_USENAME** has value **VAL_YES**, the ".rg" file generated cannot be (currently) read back.

References

[1] G. Agrawal, "Availability of Coding Based Replication Schemes," *Eleventh IEEE Symposium on Reliable Distributed Systems*, Houston, 1992.

[2] M. Ajmone Marsan, G. Balbo, and G. Conte, "A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems," *ACM Transactions on Computer Systems*, vol. 2, pp. 93–122, May 1984.

[3] J. T. Blake, A. L. Reibman, and K. S. Trivedi. Sensitivity analysis of reliability and performability measures for multiprocessor systems. In *Proc. 1988 ACM SIGMETRICS Int. Conf. on Measurement and Modeling of Computer Systems*, Santa Fe, New Mexico, 1988.

[4] Hoon Choi and K.S. Trivedi, "Approximate Performance Models of Polling Systems using Stochastic Petri Nets," *Proceedings of the IEEE INFOCOM 92*, Florence,Italy, May 4-8,1992.

[5] G. Ciardo, J. Muppala and K. Trivedi. "SPNP: Stochastic Petri Net Package," *International Conference on Petri Nets and Performance Models,* Kyoto, Japan, December 1989.

[6] G. Ciardo, J. K. Muppala, and K. S. Trivedi, "On the solution of GSPN reward models," *Performance Evaluation*, Vol. 12, No. 4, pp. 237-254, July 1991.

[7] G. Ciardo and K. S. Trivedi, "Solution of Large Generalized Stochastic Petri Net Models," in: *Numerical Solution of Markov Chains,* W. J. Stewart (ed.), Marcel Dekker, New York, 1991.

[8] G. Ciardo, J. Muppala, and K. Trivedi. Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets. *Journal of Parallel and Distributed Computing*, Vol. 15, pp. 255-269, 1992.

[9] G. Ciardo and K. S. Trivedi, "A Decomposition Approach for Stochastic Petri Net Models," *International Conference on Petri Nets and Performance Models,* Melbourne, Australia, December 1991, also *Performance Evaluation*, to appear.

[10] J. Bechta Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola, "Extended Stochastic Petri Nets: applications and analysis," in *Performance '84*, (E. Gelenbe, ed.), (Paris, France), North-Holland, Dec. 1984.

[11] R. A. Howard. *Dynamic Probabilistic Systems, Vol II: Semi-Markov and Decision Processes.* John Wiley and Sons, New York, 1971.

[12] O. Ibe and K. Trivedi, "Stochastic Petri Net Models of Polling Systems," *IEEE Journal on Selected Areas in Communications,* Vol. 8, No. 9, pp. 1649-1657, Dec. 1990.

[13] O. C. Ibe, K. S. Trivedi, A. Sathaye, and R. C. Howe, "Stochastic petri net modeling of vaxcluster system availability," in *Proceedings of the International Conference on Petri Nets and Performance Models*, (Kyoto, Japan), December 1989.

[14] O. Ibe and K. Trivedi, "Stochastic Petri Net Analysis of Finite-Population Vacation Queueing Systems," *Queueing Systems: Theory and Applications,* Vol. 8, No. 2, pp. 111-128, 1991.

[15] O. C. Ibe, H. Choi, and K. S. Trivedi. Performance Evaluation of Client-Server Systems. *IEEE Transactions on Parallel and Distributed Systems.* to appear.

[16] H. Kantz and K. Trivedi, "Reliability Modeling of the MARS System: A Case Study in the Use of Different Tools and Techniques," *International Conference on Petri Nets and Performance Models,* Melbourne, Australia, December 1991.

[17] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1978.

[18] N. Lopez-Benitez, "Dependability Analysis of Dirtsibuted Computing Systems Using Stochastic Petri Nets," *Eleventh IEEE Symposium on Reliable Distributed Systems*, Houston, 1992.

[19] J. K. Muppala, and K. S. Trivedi. "GSPN Models: Sensitivity Analysis and Applications," *Proc. 28th Annual ACM SE Region Conference,* Greenville, SC, Apr. 1990.

[20] J. K. Muppala, and K. S. Trivedi. Composite performance and availability analysis using a hierarchy of stochastic reward nets. In G. Balbo (Ed.), *Proc. Fifth Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation.* Torino, Italy, 1991, 322–336.

[21] J. K. Muppala, A. Sathaye, Rick Howe and K. S. Trivedi. Dependability Modeling of a Heterogeneous VAXcluster System Using Stochastic Reward Nets. in: *Hardware and Software Fault Tolerance in Parallel Computing Systems,* D. Averesky (ed.), Ellis Horwood Ltd., 1992, to appear.

[22] J. K. Muppala, and K. S. Trivedi. Numerical transient analysis of finite Markovian queueing systems. In Basawa and Bhat (Eds.), *Queueing and Related Models*, Oxford University Press, 1992, to appear.

[23] J. K. Muppala, S. P. Woolet, and K. S. Trivedi. Real-time systems performance in the presence of failures. *IEEE Computer.* May 1991.

[24] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.

[25] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[26] A. Reibman, R. Smith and K. Trivedi, "Markov and Markov Reward Models: A Survey of Numerical Approaches," *European Journal of Operations Research,*vol. 40, no. 2, May 1989.

[27] R. M. Smith, K. S. Trivedi, and A. V. Ramesh. Performability analysis: measures, an algorithm, and a case study. *IEEE Trans. Comput.*, C-37(4):406–417, Apr. 1988.

[28] L. Tomek and K. Trivedi, "Fixed-Point Iteration in Availability Modeling," in: *Informatik-Fachberichte, Vol. 91: Fehlertolerierende Rechensysteme,* M. Dal Cin (ed.), Springer-Verlag, Berlin, 1991.

[29] K. S. Trivedi. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1982.

[30] K. S. Trivedi, J. K. Muppala, S. P. Woolet, and B. R. Haverkort, "Composite performance and dependability analysis," *Performance Evaluation*, vol. 14, no. 3-4, pp. 197–215, 1992.

APPENDIX: VMS installation

The version running under the VMS operating system uses exactly the same source code, hence it has the same features. Differences exist at the command language level, though. First of all, two files ".COM" are provided, instead of "makefiles".

The first file, SPNPINST.COM, is used to compile all the C files composing the package. Since it does not use the "make" facility, all the files are recompiled, even if some of them do not need to be recompiled. Type `@SPNPINST` to execute it.

The second file, SPNP.COM, is used to run the package. It requires a filename without extension, as in the UNIX version. For example `@SPNP TEST` will run the package on the CSPL file TEST.C, which must exist. The interaction that follows is slightly different from the one in UNIX, again because the "make" facility is not used. First of all, if a filename is not specified, a message will prompt the user to input one. Then the user is asked whether the compilation of the CSPL has to take place or not, to avoid needlessly repeating it. If, for example, the user wants to run again the SRN with different run-time parameters, the compilation does not have to be performed again, only analysis phase must be run.

A third file, SPNPTEST.COM, is provided, to run the examples provided with the package (EXAMPLE1.C, ...) on predefined input files (EXAMPLE1.INP, ...).