SPNP Users Manual
Version 4.0

Gianfranco Ciardo
Ricardo M. Fricks
Jogesh K. Muppala
Kishor S. Trivedi

March 15, 1994

Contact: Kishor S. Trivedi
Department of Electrical Engineering
Duke University – PO Box 90291
Durham, NC – 27708-0291
(919) 660 - 5269
(919) 493 - 6563
kst@ee.duke.edu

Abstract

We present the **Stochastic Petri Net Package** (SPNP), a versatile modeling tool for performance, dependability, and performability analysis of complex systems. Input models developed based on the theory of stochastic reward nets are solved by efficient and numerically stable algorithms. Steady-state, transient, cumulative transient, time-averaged, and "up-to-absorption" measures can be computed. Parametric sensitivity analysis of these measures is possible. Some degree of logical analysis capabilities are also available in the form of assertion checking and the number and types of markings in the reachability graph. Advanced constructs available - such as marking-dependent arc multiplicities, guards, arrays of places and transitions, and subnets - reduce modeling complexity and enhance power of expressiveness of the package. The most powerful feature is the capability to assign reward rates at the net level and subsequently compute the desired measures of the system being modeled. The modeling description language is CSPL, a C-like language, although no previous knowledge of the C language is necessary to use SPNP.

1 ABOUT THIS MANUAL

This manual[1] describes Version 4.0 of SPNP, 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; **Appendix A** contains some minor differences for the VMS version.

A basic knowledge of the **Stochastic Reward Net** (SRN) [???] formalism and **Markov chains** is assumed [29]. The SPN model we adopt is best described in [5, 6] [Ciardo, Blakemore et al.], but it may be useful to consult [1]. The reader should consult [2, 23, 24, 25] if unfamiliar with some **Petri Net** (PN) concepts. For further information on Markov chains, performance modeling and reliability modeling see [29], while for performability modeling see [27, 30] [May 91 IEEE Computer and the Chapter in Krishna&Lee]. Markov and Markov reward model solution techniques are surveyed in [26]. 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.

In the next section, we list some of the major features of the package. In the subsequent section we explain how to describe a SRN to the package. The structure of the package and

[1] This manual was composed mainly based on texts extracted from [5, 10].

format of the files generated by SPNP is then presented, followed by the list of the available options. In the conclusion we list future work planned on the package.

2 WHAT IS NEW IN SPNP VERSION 4.0

The salient features of SPNP Version 4.0 as it differs from SPNP Version 3.1 are:

- A new transient uniformization method which uses Fox-Glynn method for computing the poisson probabilities has been implemented for the instantaneous probability computation. The new method can be invoked by setting IOP_METHOD to VAL_FOXUNIF.
- You can switch off steady-state detection in transient analysis by using the new switch IOP_SSDETECT. Set it to VAL_YES for detection and VAL_NO for no detection. The default is VAL_YES.
- The former "enabling" functions have now been renamed to "guard". For backward compatibility, a flag has been defined so that previous CSPL files can still be run with the new SPNP 4.0.
- We can now set time to INFINITY while doing transient analysis. The code will recognize that you wish to compute steady-state solution and does so automatically. Use new flag IOP_SSMETHOD to specify which steady-state solution method you wish to use.
- Implemented the POWER method for steady-state computation. This method is guaranteed to converge while no such guarantee is available for SOR or Gauss-Seidel [Goyal, Lavenberg & Trivedi paper]. On most problems, however, SOR will work and will converge much faster. Therefore, power method should be considered as a backup. To invoke power method, set IOP_METHOD to VAL_POWER.

3 THE SPNP PACKAGE

Reliability block diagrams and fault trees are commonly used for system *dependability analysis* (reliability/availability). These model types allow a concise description of the system under study and can be evaluated efficiently, but they cannot represent dependencies

ocurring in real systems. Markov models, on the other hand, are capable of capturing various kinds of dependencies that occur in reliability/availability models (these dependability model types are compared in [Malhotra & Trivedi, to appear]).

Task precedence graphs can be used for *performance analysis* of concurrent programs with unlimited system resources. System performance analysis using product-form queueing networks can instead consider contention for system resources. The product-form assumptions are not satisfied, however, when behaviors such as concurrency within a job, synchronization, and server failures are considered. Once again, Markov models do provide a framework to address all these concerns.

Traditionally, performance analysis assumes a fault-free system. Separately, dependability analysis is carried out to study system behavior in the presence of component faults, disregarding different performance levels in different configurations. Several different types of interactions and corresponding tradeoffs have prompted researchers to consider the combined evaluation of performance and dependability. Most work on the combined evaluation (sometimes called *performability analysis*) is based on the extension of Markov processes to **Markov reward processes** [11, 27] [Smith & Trivedi chapter in Takagi book], where a reward is attached to each state of the Markov process.

Markov reward processes have the potential to reflect concurrency, contention, fault-tolerance, and degradable performance. They can be used to obtain not only program/system performance and system reliability/availability measures, but also combined measures of performability.

The common solution for modeling dependability, performance, or performability would then appear to be the use of Markov reward models, but one major drawback of Markov reward models is the largeness of their state space. To address this problem we designed SPNP, a versatile modeling tool for solution of SRN models [5, 6]. The SRN model can be used to generate a (large) underlying Markov reward model automatically starting from a concise description.

4 SPNP FEATURES

The input language for SPNP is CSPL (**C**–*based* **S***tochastic* **P***etri net* **L***anguage*). A CSPL file is a C file [17]; it is compiled using the C compiler and then linked with the precompiled files constituting SPNP. The full expressive power of the C programming language is available to increase the flexibility of the net description. Being a superset of the C language is an

important feature of CSPL since its user can exploit C language constructs to represent a large class of SRNs within a single CSPL file. Although, most applications will only require a limited knowledge of the C syntax, since predefined functions are available to define SPNP objects.

Other important characteristic of SPNP is the provision of a function to input a value at runtime, before reading the specification of the SRN. This input value can be used in SPNP to modify the value of a scalar parameter as well as the structure of the SRN itself. Arrays of places or transitions and subnets are two features of the package useful to exploit this *structural parametrization*. A single CSPL file is sufficient to describe any legal SRN, since the user of SPNP can input at run-time the number of places and transitions, the arcs among them, and any other required parameter. In practice, the class of SRNs described by a single CSPL file is more likely to represent only minor variations on a common structure, so as to increase compactness and consistency. A single file can be used to represent all the SRNs corresponding to a given system under consideration, even when they differ somewhat in their structure. As a result, the numerical parameters used in the specification of rates and probabilities need to appear in only one file, decreasing the risk of having inconsistent definitions in different files.

The SPNP package allows the user to perform $steady-state$, $transient$, $cumulative\,transient$, and *sensitivity analysis* of SRNs. A review of the solution techniques utilized is presented in [5] [Ciardo,Blakemore et al.]. Steady-state analysis is often adequate to study the performance of a system, but time-dependent behavior (transient analysis) is sometimes of greater interest: instantaneous availability, interval availability, and reliability (for a fault-tolerant system); response time distribution of a program (for performance evaluation of software); computational availability (for a degradable system) are some examples. Sensitivity analysis is useful to estimate how the output measures are affected by variations in the value of input parameters, allowing the evaluation of alternatives during the design phase of a system. Other important applications of sensitivity analysis are system optimization and bottleneck analysis [Blake, Ribman & Trivedi].

Sophisticated steady-state and transient solvers are available in the SPNP package; cumulative and up-to-absorption measures can be computed. In addition, the user is not limited to a predefined set of measures: detailed expressions reflecting exactly the measures sought can be easily specified. The measures are defined in terms of reward rates associated with the markings of the SRN. The analytic solution methods provided in the package address the $stiffness$ problems that is often encountered in reliability and performance models.

A number of important Petri net constructs like marking dependency, variable cardinality arc, guards, arrays of places and transitions, subnets, and assertions [5] [Ciardo, Blakemore

et al.] facilitate the construction and debugging of models for complex systems.

Although model hierarchies are not built into SPNP, hierarchical SRN models can be exercised using a UNIX shell script 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, 8, 16, 19, 21, 22, 28]. Also, the user can consult several papers that have appeared in the literature where SPNP was used [4, 7, 9, 8, 13, 12, 14, 15, 16, 19, 22, 21, 28] [IEEE-TSE Nov 93, Robertazi; Proc IEEE Jan 94, Agrarwal & Noe, ...]. See [Malhotra & Trivedi] for the importance of reward definition at the net level.

5 THE CSPL LANGUAGE

Modeling with the SPNP package implies that an input file describing the system structure and behavior must be written. The language designed to do so is named **CSPL**, a superset of the **C** language [17]. What distinguishes CSPL from C is a set of predefined functions specially developed for the description of SRN entities. Any legal C construct can be used anywhere in the CSPL file. All the C library functions, such as **fprintf**, **fscanf**, **log**, **exp**, etc., are available and perform as expected. The only restriction to this generic rule is that the file should not have a **main** function.

In spite of being a programming language, CSPL enables the user to describe SRN models very easily. To help a new user, example input files are provided in the accompanying **Reference Guide** (i.e., **example1.c**, **example2.c**, etc.). There is no need to be a programmer to fully exploit all the built-in features of the SPNP package. Just a basic knowledge of C is sufficient to describe SRNs effectively. Although, for experienced programmers CSPL brings the full power and generality of the C language.

A CSPL input file has the following basic structure[2]:

```
/* begin of CSPL file */
parameters () {
  ......
}
net() {
  ......
```

[2] All the six functions listed must be present in the CSPL file, even if their contents happen to be empty.

```
}
assert() {
  ......
}
ac_init() {
  ......
}
ac_reach() {
  ......
}
ac_final() {
  ......
}
/* end of CSPL file */
```

Figure 1 shows the net for the running example that will be used to illustrate the syntax and semantics of the functions to be presented[3]. Besides presenting the functions, the example will be developed so as to suggest a general methodology of describing models using CSPL.

5.1 Parameters Function

The function **parameters** allows the user to customize the package. Several parameters establishing a specific behavior can be selected (the whole set of parameters and default values are visualized in table 1[4]).

The options are selected by means of the following function calls:

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

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

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

[3]The whole CSPL file describing the example is presented in **Appendix B**
[4]For convenience the semantics of all the options available are collectively described in **Appendix C**.

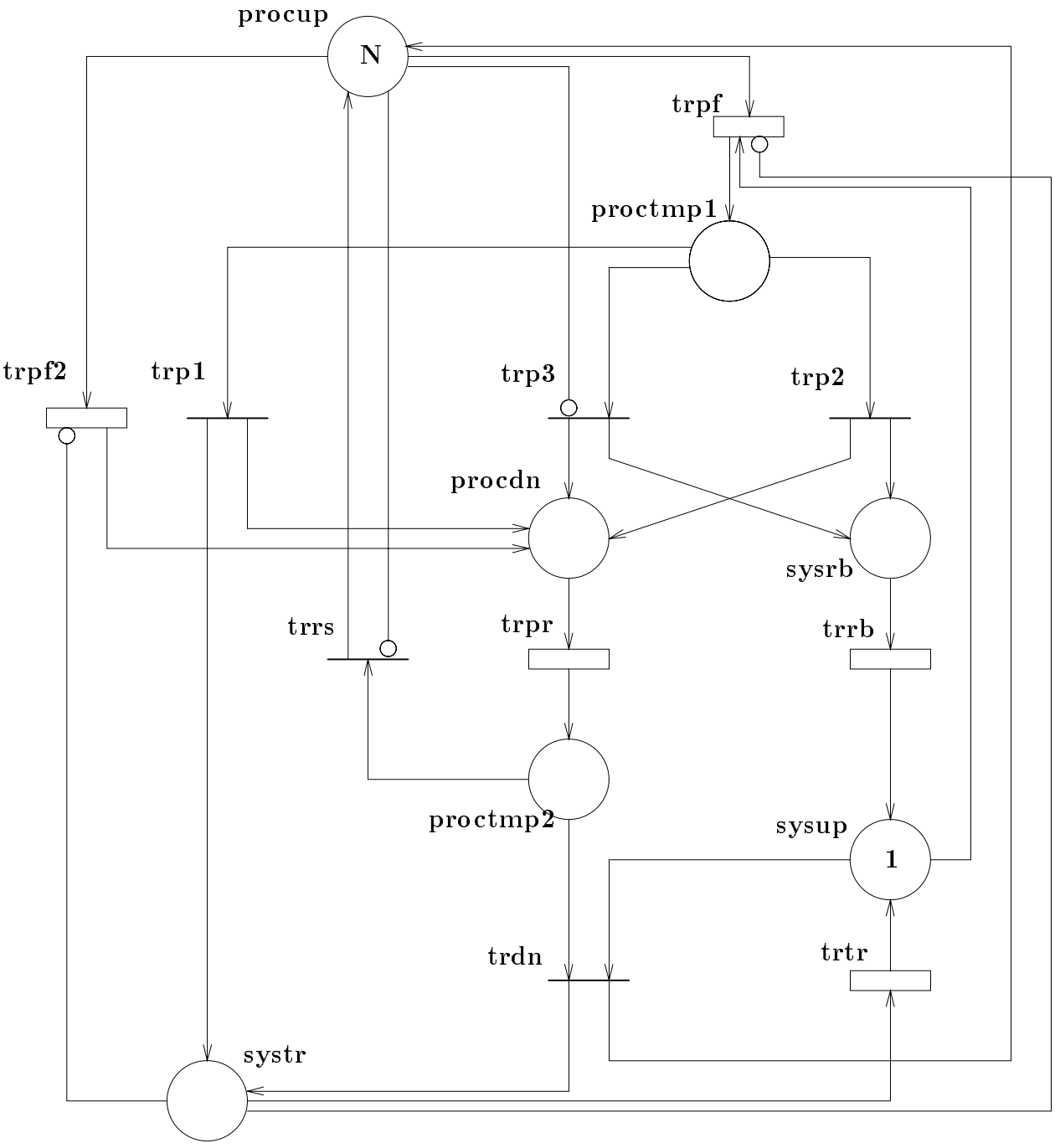

Figure 1: A generic Stochastic Reward Net

type	name	values	default
int	IOP_CUMULATIVE	VAL_YES; VAL_NO	VAL_YES
int	IOP_DEBUG	VAL_YES; VAL_NO	VAL_NO
int	IOP_ITERATIONS	non-negative **int**	2000
int	IOP_MC	VAL_CTMC; VAL_DTMC	VAL_CTMC
int	IOP_METHOD	VAL_SSSOR; VAL_GASEI;	
		VAL_TSUNIF; VAL_FOXUNIF;	
		VAL_POWER	VAL_SSSOR
int	IOP_OK_ABSMARK	VAL_YES; VAL_NO	VAL_NO
int	IOP_OK_TRANS_M0	VAL_YES; VAL_NO	VAL_YES
int	IOP_OK_VANLOOP	VAL_YES; VAL_NO	VAL_NO
int	IOP_PR_FULL_MARK	VAL_YES; VAL_NO	VAL_NO
int	IOP_PR_RGRAPH	VAL_YES; VAL_NO	VAL_NO
int	IOP_PR_RSET	VAL_YES; VAL_NO;	
		VAL_TANGIBLE	VAL_NO
int	IOP_PR_MARK_ORDER	VAL_CANONIC; VAL_LEXICAL;	
		VAL_MATRIX	VAL_CANONIC
int	IOP_PR_MC	VAL_YES; VAL_NO	VAL_NO
int	IOP_PR_MC_ORDER	VAL_FROMTO; VAL_TOFROM	VAL_FROMTO
int	IOP_PR_MERG_MARK	VAL_YES; VAL_NO	VAL_YES
int	IOP_PR_PROB	VAL_YES; VAL_NO	VAL_NO
int	IOP_SENSITIVITY	VAL_YES; VAL_NO	VAL_NO
int	IOP_SSDETECT	VAL_YES; VAL_NO	VAL_YES
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 for the parameters function

The function **iopt** (**fopt**) enables the user to set *option* to have the integer (double-precision floating point) *value*. Any of the available options (listed in table 1) can be selected and modified. The user should always be sure to pass to these functions 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 **Appendix C**), instead of the default canonical order, the order in which they are found.

The function **input** permits the input of parameters at run time, i.e., when **parameters** is being executed. A message of the form **Please type** "*msg*" is displayed on the **stderr** stream (usually assigned to the console), and the program waits for a double-precision floating point value from the **stdin** stream (usually assigned to the keyboard). The returned value is a **double**, so an explicit type conversion to **int** or **double** may be needed. The assigned value is printed in the ".out" file, together with *msg*. This feature is useful to relate the set of options selected to a particular running of a CSPL program.

For our example of Figure 1 let us assume the following **parameters** function:

```
int    N, /* total number of processors in the cluster */
       q; /* number of processors needed for quorum    */
double c; /* coverage factor */

parameters() {
  iopt(IOP_METHOD,VAL_TSUNIF);
  iopt(IOP_PR_FULL_MARK,VAL_YES);
  iopt(IOP_PR_MC,VAL_YES);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_PR_RSET,VAL_YES);
  fopt(FOP_PRECISION,0.00000001);
```

```
    N = input("Enter the total number of processors");
    q = input("Enter number of processors needed for quorum");
    c = input("Enter the coverage factor");
  }
```

5.2 Net Function

The function **net** allows the user to completely define the structure and parameters of a SRN model. The functions that can be used inside the function **net** will be explained in the following, grouped according their logical function in the language.

5.2.1 Specifying a Petri Net

The standard Petri net (PN) model [24] is defined by a set of *places* (drawn as circles), a set of *transitions* (drawn as bars), and a set of *directed arcs*, which connect transitions to places or places to transitions (see Figure 1). Places may contain *tokens* (represented by **integers**). The state of the PN, called the *PN marking*, is defined by a vector enumerating the number of tokens in each place.

The states of a PN can be used to represent various entities associated with a system - for example, the number of functioning resources of each type, the number of tasks of each type waiting at a resource, the number of concurrently executing tasks of a job, the allocations of resources to tasks, and states of recovery for each failed resource. Transitions represent the changes of states due to the ocurrences of simple or coumpound events such as the failure of one or more resources, the completion of executing tasks, or the arrival of jobs.

A place is an input to a transition if an arc exists from the place to the transition. If an arc exists from a transition to a place, it is an output place of the transition. A transition is *enabled* when each of its input places contains at least one token. Enabled transitions can fire, by removing one token from each input place and placing one token in each output place. Thus the firing of a transition may cause a change of state (producing a different marking) of the PN. The *reachability set* is the set of markings that are reachable from a given *initial marking*. The reachability set together with arcs joining the marking indicating the transition that cause the change in marking is called the *reachability graph* of the net.

The functions listed below allow the user to describe Petri nets:

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

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

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

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

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

The function **place** (**trans**) defines a place (transition) with identifier *name*. All names must be distinct, i.e., the places/transitions in the Petri net should have distinct names[5]. A place/transition name is legal if:

- its length is between 1 and **MAXNAMELENGTH**, as defined in the file **const.h**[6], by default this constant has the value 20;
- it is composed of the characters {**0..9,a..z,A..Z,_**} only;
- the first character is in {**a..z,A..Z**}.

If an arc is directed from a place *p_name* to a transition *t_name* then it is an *input arc* to the transition and is described by the function **iarc**. If an arc is directed from a transition *t_name* to a place *p_name* then it is an *output arc* to the transition and is described by the function **oarc**.

The distribution of tokens in a marked Petri net defines the state of the net and is called its *marking*. The initial marking of the Petri net is created by successive use of the function **init**. With this function it is possible to initialize the number *n* of tokens in any place identified by its *name*. By default, places are otherwise initially empty (zero tokens).

[5] 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).

[6] 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.

As an illustration of the use of the first group of functions, we describe the underlying Petri net of Figure 1 (starting from top to bottom and from left to right) by the following segment of code:

```
net() {

  /* places: */          /* initial markings: */
  place("procup");       init("procup",N);
  place("proctmp1");
  place("procdn");
  place("sysrb");
  place("proctmp2");
  place("sysup");        init("sysup",1);
  place("systr");

  /* transitions: */     /* input arcs: */           /* output arcs: */
  trans("trpf");         iarc("trpf","procup");      oarc("trpf","proctmp1");
                         iarc("trpf","sysup");
  trans("trpf2");        iarc("trpf2","procup");     oarc("trpf2","procdn");
  trans("trp1");         iarc("trp1","proctmp1");    oarc("trp1","systr");
                                                     oarc("trp1","procdn");
  trans("trp3");         iarc("trp3","proctmp1");    oarc("trp3","procdn");
                                                     oarc("trp3","s ysrb");
  trans("trp2");         iarc("trp2","proctmp1");    oarc("trp2","procdn");
                                                     oarc("trp2","sysrb");
  trans("trrs");         iarc("trrs","proctmp2");    oarc("trrs","procup");
  trans("trpr");         iarc("trpr","procdn");      oarc("trpr","proctmp2");
  trans("trrb");         iarc("trrb","sysrb");       oarc("trrb","sysup");
  trans("trdn");         iarc("trdn","proctmp2");    oarc("trdn","systr");
                         iarc("trdn","sysup");       oarc("trdn","procup");
  trans("trtr");         iarc("trtr","systr");       oarc("trtr","sysup");

  ..... /* other functions to be presented */

}
```

where N is defined elsewhere outside the **net** function.

After the original conception, some extensions of the original theory of PNs were proposed.

The concept of *multiple arcs* was introducted: input and output arcs may have an associated *multiplicity* factor. An arc with a multiplicity factor k is logically equivalent to k parallel arcs. This construct increases the convenience of use of the PNs by allowing multiple arcs between transitions and places to be represented in a compact form.

Inhibitor arcs, on the other hand, increase the fundamental modeling power or decision power of ordinary Petri nets. An inhibitor arc from a place to a transition has a circle rather than an arrowhead at the transition. The firing rule for the transition is changed such that the transition is *disabled* if there is at least one token present in the corresponding inhibiting input place[7].

The concept of arc multiplicity can also be applied to inhibitor arcs: an inhibitor arc with multiplicity k needs at least k tokens in the corresponding input place to *disable* the transition.

Therefore, the following group of functions was included in the SPNP to add more power and compactness to the PN models:

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*;

The function **harc** allows the user to define inhibitor arcs from transition *t_name* to place *p_name*. The functions **miarc**, **moarc**, and **mharc** define multiple input, output, and inhibtor arcs, respectively, with multiplicity *mult* (a positive integer constant). Intuitively, a multiple arc with multiplicity *mult* can be thought of as *mult* arcs having the same source and destination. An inhibtor arc from place p to transition t with multiplicity m will disable t in any marking where p contains at least m tokens.

[7] No tokens are moved along inhibitor arcs.

We can now expand our previous description of Figure 1 with the following code:

```
net() {

  ..... /* previously presented functions */

  /* inhibitor arcs: */
  harc("trpf","systr");
  harc("trpf2","systr");

  /* multiple inhibitor arcs: */
  mharc("trp3","procup",q);
  mharc("trrs","procup",q);

  ..... /* other functions to be presented */

}
```

5.2.2 Guards and Priorities

There are a few additional extensions that are included in the package for the sake of user convenience in defining more complex enabling and inhibiting conditions. The first extension to be presented is the concept of *guard*, which attach logical conditioning functions or enabling functions to the transitions.

A user can define conditioning functions involving relational and numerical operators of the number of tokens in various places and the enabled state of a transition. Such guards are often used to reduce the graphical complexity of the Petri net. Guards can also used to specify *state truncation* [5].

Still another way to define inhibitions on the net is provided. A *priority level* may be defined for each transition in the net, so that the set of transitions is partitioned by priority levels. The firing rule is then modified such that a transition is inhibited if there is a higher priority transition enabled, while the firing of transitions with the same priority level follows the standard rules.

Furthermore, priorities were also included in the package because we may often wish to model the probabilistic outcome of an event represented by the firing of a 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[8].

If we ignore timing, we can imagine an ordinary PN as a SRN where all transitions have the same priority, where no inhibitor arcs are present, and where the guards are identically equal to 1 ($TRUE$). We also can observe that guards and priorities enhance the power of PNs by extending the firing rules: Each transition t has an associated (boolean) guard e[9]. The function is evaluated in marking M when "there is a possibility that t is enabled," that is, when:

- no transition with priority higher than t is enabled in M;
- the number of tokens in each of its input places is larger than or equal to the (variable) cardinality of the corresponding input arc;
- the number of tokens in each of its inhibitor places is less than the (variable) cardinality of the corresponding inhibitor arc. Only the $e(M)$ is evaluated; t is declared enabled in M iff $e(M) = TRUE$.

The ability to express complex enabling/disabling conditions textually is invaluable. Without it the modeler might have to add extraneous arcs or even places and transitions to the SRN, to obtain the desired behavior.

The functions listed below provide additional constructs to selectively disable a transition in a marking which would otherwise enable it:

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

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

[8]The use of priorities provide another way to disable a transition.

[9]By default transitions have the lowest priority (zero) and have no guards (their enabling function are set to the constant $TRUE$).

The function **guard** (**priority**) defines the enabling function (priority) for transition *name* to be *efunc* (*prio*)[10]. The function passed as actual parameter to the guards must be defined in the CSPL file before being used.

To further expand the description of the model presented in Figure 1 we write:

```
net() {

  ..... /* previously presented functions */

  /* guards: */
  guard("trpf2",ensysr);
  guard("trp1",enrstr);
  guard("trp2",enrstr);
  guard("trpr",enprcrp);
  guard("trrb",enrstr);
  guard("trtr",enrstr);

  /* priorities: */
  priority("trp1",10);
  priority("trp3",10);
  priority("trp2",10);
  priority("trrs",10);
  priority("trdn",10);

  ..... /* other functions to be presented */

}
```

where the enabling functions **ensysr**, **enrstr**, and **enprcrp** are described elsewhere outside the **net** function.

[10]The following types are predefined, for the purpose of clarity: **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).

5.2.3 Probabilistic Behavior

We may often wish to model the probabilistic outcome of an event represented by the firing of a transition. SPNP provides *random switches* [1] for this purpose. A random switch consists of a set of *immediate transitions* together with a set of (possibly marking dependent) weights.

Whenever a subset of transitions controlled by a random switch is enabled, a probability of firing (normalized so that they sum to the unity) is defined for each enabled transition. The particular transition to fire is then chosen according to this (discrete) probability distribution. We require that the sets of transitions for each random switch be disjoint, thus we can further assign a unique priority level to each random switch. This prioritizing of random switches defines the outcome when transitions belonging to two random switches are simultaneously enabled. (Any enabled transition that does not belong to a random switch will be considered a random switch with one element). The choice of the random switch is made according to the priority level of the switch; the choice of the transition within the chosen random switch is made as if it were the only one active.

The following group of functions allows the description of random switches:

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*)();

The functions **probval**, **probdep**, and **probfun** allow the assignment of firing probabilities to transitions. These functions define respectively the probability of transition *name* as a constant value *val* of type **probability_type**, 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 **probability_type**.

Immediate transitions are drawn as thin bars in our running example (Figure 1) and their

firing probabilities are described by the following code:

```
net() {

  ..... /* previously presented functions */

  /* firing probabilities of immediate transitions: */
  probval("trp1",c);
  probval("trp3",1.0);
  probval("trp2",1.0-c);
  probval("trrs"1.0);
  probval("trdn",1.0);

  ..... /* other functions to be presented */

}
```

where c is defined elsewhere outside the **net** function. An error occurs 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 an error is not caught by the C compiler (**spnp** command), but it is caught by **lint** (**spnpcheck** command).

5.2.4 Timing of Events

Transitions in SPNP are allowed to belong to two different classes: *immediate* and *timed*. Immediate transitions have higher priority than timed transitions and once enabled fire immediately. To the timed transitions are assigned random variables, called *firing times*, characterized by a given cumulative distribution function[11]. The firing time represents the time needed to complete the activity associated with the transition in a specific marking. Therefore, once a timed transition is enabled, an random amount of time elapses. After this delay, if the transition is still enabled, it will then fire.

If more than one transition is enabled in a given marking we need to define a set of rules for choosing the one to fire. If there is an immediate transition among those enabled, it fires first, with probability one. If more than one immediate transition is simultaneously enabled,

[11] Only exponential distribution is allowed currently.

the one to fire is chosen according to random switch(es) that contain the transitions. If there are only timed transitions enabled the most logical choice for the one to fire is that with the minimum firing time.

When specifying transitions the following rules should be considered:

- each defined transition must have one (and only one) function assigning a rate (timed transitions), or a probability (immediate transitions) to it;
- transitions cannot be disabled by defining rates or probabilities that evaluate to zero (0) in the marking. Instead, the user must explicitly disable transitions;
- the package exits with an error message if a non-positive rate or probability is found for a transition which would be otherwise enabled.

Rates of timed transitions are defined using the following functions calls:

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*)();

The functions **rateval**, **ratedep**, and **ratefun** allow the assignment of rates to transitions. These functions define respectively the rate of transition *name* as a constant value *val* of type **rate_type**, 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**.

To illustrate the strength of marking dependency (function **ratedep**) let us consider the CSPL segment below:

```
ratefun_type myval() {return(mark("left_place")*7.3);}
```

```
net() {
  place("left_place");
  place("right_place");
  trans("from_left_to_right");
  trans("from_right_to_left");
  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");
  init("left_place",4);
  ratefun("from_left_to_right",myval);
  rateval("from_right_to_left",1.0);
}
```

The **net** function defines an 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

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

which would have been more efficient, and perhaps clearer. However, **ratefun** allows more general marking dependency compared with **ratedep**.

In our running example (Figure 1) timed transitions are drawn as rectangles and their firing rules are described by the following code:

```
net() {

  ..... /* previously presented functions */

  /* timed transitions: */
  rateval("trpr",1.0/2.0);
  rateval("trrb",6.0);
  rateval("trtr",120.0);
```

```
    ratedep("trpf",1.0/5000.0,"procup");
    ratedep("trpf2",1.0/5000.0,"procup");

    ..... /* other functions to be presented */

}
```

5.2.5 Variable Cardinality Arcs

SPNP allows variable cardinality input, output, and inhibitor arcs.

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 (*$func$)();

The syntax to express marking dependency is a natural extension of the syntax to describe marking independent behavior. The functions **viarc**, **voarc**, and **vharc**[12] define respectively an input, output, and inhibitor arc from transition t_name to place p_name, with multiplicity given by the marking dependent function $func$ (returning an **integer**, which represents the cardinality of the arc[13]).

A typical example of the power of expression of this construct 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*.

[12] We chose to define **vharc** for completeness, but it is usually more efficient to use a guard instead.
[13] When the cardinality of the arc is zero, the arc is considered absent.

Perhaps even more importantly, if t is a timed transition, 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. The variable cardinality should be used only when really needed, because it may make the SRN harder to understand and it requires more computation (in the reachability graph generation) that a standard or multiple arc.

Using the concept of marking dependent arc cardinalities associated with timed transitions might give rise to nonintuitive or unforeseen behaviors; for example, in the flushing of tokens just described, t is enabled in any marking, even when p is empty, unless:

- other input arcs are defined for t;
- an enabling function is used to explicitly disable t when p is empty and possibly in other cases as well; or
- 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).

5.2.6 Modeling Marking Dependence

Perhaps the most important characteristic of CSPL is the ability to allow extensive marking dependence. Parameters such as the rate of a timed transition, the cardinality of an input arc, or the reward rate in a marking, can be specified as complex functions of the number of tokens in some (possibly all) places. Marking dependence can lead to more compact models of systems. Very complex behavior can be modeled in this way; the complexity is reflected in the way in which places are combined together.

The following functions are available to specify marking dependence:

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

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

When called the function **mark** returns the number of tokens in place *p_name*. The function **enabled** returns 1 ($TRUE$) if transition *t_name* is enabled, 0 ($FALSE$) otherwise.

Using these functions we can use the full power of C to devise complex marking dependent functions. For example, if the rate of transition $t1$ equals the number of tokens in place $p1$ time $MU1$ (a constant) plus the number of tokens in place $p2$ times $MU2$ (another constant), we can define the marking dependent function

```
probability_type rate1() {
  return(mark("p1")*MU1 + mark("p2")*MU2);
}
```

and then use it in the specification of the rate for transition $t1$:

```
trans("t1"); ratefun("t1",rate1);
```

To further expand the description of the model presented in Figure 1 we must write:

```
/* guards: */
int ensysr()  {return(mark("sysrb");}
int enrstr()  {return(mark("procup")>=q ? 1:0);}
int enprcrp() {
  return(mark("procup")>=q && mark("systr")+mark("sysbr")>0 ? 0:1);
}

net() {

  ......

}
```

5.2.7 Arrays of Places and Transitions

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).

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:

```
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 result 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 indeces 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 relates 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 context: **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.

5.2.8 Sensitivity Analysis

Sensitivity analysis is useful to estimate how the output measures of a system model are affected by variations of its input parameters. It is then very attractive to evaluate alternatives during the design phase of a system, when exact values for all the input parameters may not be known yet. Other important applications of sensitivity analysis are system optimization and bottleneck analysis. An SRN model lends itself to sensitivity analysis at various levels. The SPNP package offers the possibility of computing sensitivity measures by analyzing the underlying Markov process. Sensitivity analysis of Markov and Markov reward models is discussed in [3] and the sensitivity analysis of SRN models is discussed in [18].

When sensitivity analysis is needed, both the probability function (rate) and its derivative need to be specified:

PROBABILITY DERIVATIVE:

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*)();

RATE DERIVATIVE:

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*)();

These functions specify the derivative of probability functions (rates), and are exclusive with the others functions describing the properties of transitions (i.e., you cannot use both simultaneously to describe the same transition).

5.3 Assert Function

The function **assert** allows the evaluation of a logical condition on a marking of the SRN[14]. **Assert** is especially useful with large and complex SRNs, where it greatly reduces the time to discover simple errors, such as a missing arc or an incorrect cardinality specification.

[14]The check on the legality of a marking is performed using the same functions used to achieve marking dependency, namely **mark** and **enabled**.

The fundamental principle is to verify general assertion(s) that are believed to hold in any marking of the reachability set.

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 or RES_NOERR if the marking is (thought to be) legal[15]. If RES_ERROR is returned, the execution then halts, the present marking is displayed, and the partially generated reachability graph is written to a file for debugging purposes. If the illegal marking is caused by an unforeseen sequence of transition firings, finding that sequence using the output information is usually a fast process even in large reachability graphs (tens of thousands of markings). This check is turned off by setting the function identically equal to RES_NOERR.

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 behavior.

For example, the **assert** definition

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

This type of check is limited, since it helps detect the presence of illegal markings or firing sequences, but it cannot detect the absence of legal markings or 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 bounded nets. The examination of the whole (infinite) reachability graph is out of the question since the program will terminate printing a message for insufficient memory.

For the running example (Figure 1) we devised the following **assert** function:

[15] RES_ERROR and RES_NOERR are predefined values

```
assert() {return(RES_NOERR);}
```

5.4 Ac_init Function

The function **ac_init** is called before starting the reachability graph construction. The function **pr_net_info** can be used in it, to output data about the SRN in the ".out" file (see section about intermediate files). 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).

For the example of Figure 1 we wrote the following **ac_init** function:

```
ac_init() {fprintf(stderr,"\nProcessor Cluster Model\n\n");}
```

5.5 Ac_reach Function

The function **ac_reach** is called after the reachability graph construction is completed. The function **pr_rg_info** can be used in it, to output data about the reachability graph in the ".out" file (this does not affect the generation of the ".rg" file).

For the running example (Figure 1) we left this function empty:

```
ac_reach() {}
```

5.6 Ac_final Function

The function **ac_final** is called after the solution of the **Continuous Time Markov Chain** (CTMC) has completed, to allow user-requested outputs. Table 5.6 presents a list of generic functions that can be used inside the **ac_final** function.

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

name and syntax	behavior
pr_mc_info();	Output of data about the CTMC and its solution.
set_prob_init(*fnc*) reward_type (**fnc*)();	Allows the user to define the initial probability vector over the markings of the SRN.
rate_type **rate**(*t_name*) char **t_name*;	Express the marking dependent rate of transition *t_name* (defined as 0 when the transition is not enabled in the marking).
pr_value(*str*,*expr*) char **string*; double *expr*;	Prints the string *str* and the value of expression *expr*.
pr_message(*str*) char **str*;	Allows the user to print an arbitrary message (*str*) in ".out" file.

Table 2: Generic functions to be used inside the ac_final function

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 **set_prob_init**. 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 expression to be printed could be any one 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
```

In addition, desired output measures of the transient analysis may be requested in this function. Table 5.6 lists predefined functions available in SPNP to print information about each place and transition, or to allow the specification of user-defined measures.

name and syntax	outputs (written on the ".out" file)
pr_std_average();	For each place: . probability that it is not empty; . its average number of tokens. For each timed transition: . probability that it is enabled; . its average throughput.
pr_std_average_der();	Derivatives of all the above standard measures.
pr_expected(*str*,*fnc*) char **str*; reward_type (**fnc*)();	The string *str* and the expected value of function *fnc*.
pr_der_expected(*str*,*fnc*,*dfnc*) char **str*; reward_type (**fnc*)(),(**dfnc*)();	The string *str* and the derivative *dfnc* of the expected value of the function *fnc* with respect to the parameter θ.
pr_sens_expected(*str*,*fnc*,*dfnc*) char **str*; reward_type (**fnc*)(),(**dfnc*)();	The string *str*, the expected value of the function *fnc*, and its derivative *dfnc* with respect to the parameter θ.
reward_type **expected**(*fnc*) reward_type (**fnc*)();	The expected value of function *fnc*.
reward_type **der_expected**(*fnc*,*dfnc*) reward_type (**fnc*)(),(**dfnc*)();	The derivative *dfnc* of the expected value of the function *fnc* with respect to the parameter θ.

Table 3: Available options for especification of output measures for steady state analysis

COMMENTS:

- 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.

- *fnc* should be a marking dependent reward function.

- **pr_std_average_der** and **pr_der_expected** can be used iff steady-state sensitivity is performed.

Examples of specifications of output measures:

- The function

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

  will print on the ".out" file:

  ```
  EXPECTED: utilization = 3.2
  ```

  if the expected value of the reward function *util*[16] is 3.2.

- 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:

time_value(t)
double t;

[16] 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.

name and syntax	outputs (written on the ".out" file)
pr_time_avg_expected(*fnc*) reward_type (**fnc*)();	The time-averaged expected value of function *fnc*.
pr_mtta(*str*) char **str*;	The string *str* and the mean time to absorption for the SRN. This function should be used only when the underlying CTMC has absorving states.
pr_cum_abs(*str*,*fnc*) char **str*; reward_type (**fnc*)();	The string *str* and the expected accumulated reward until absorption for a CTMC with absorbing states. To use this function the corresponding reward rate should be specified
cum_abs(*fnc*) reward_type (**fnc*)();	The expected accumulated reward until absorption.

Table 4: Available options for especification of output measures for transient analysis

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 specified time t, 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; and
- **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.

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 following function **ac_final** was written for the running example (Figure 1):

```
reward_type availab() {return(mark("sysup"));}
reward_type ppower()  {return(mark("procup")*mark("sysup"));}

ac_final() {
  int i;

  for(i = 1; i <= 10, i++) {
    time_value((double)i);
    pr_expected("Availability",availab);
    pr_time_avg_expected("Availability",availab);
    pr_expected("Processing Power",ppower);
    pr_cum_expected("Processing Power",ppower);
  }
}
```

5.7 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 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, 8].

The specialized set of output functions provided by SPNP are:

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

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

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

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

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

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

Accumulated and **pr_accumulated** 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 linear system having as many variables as the non-zero-reward markings, so it can be expensive.

Set_prob0 and **scale_prob0** are used to set the initial state probability for the computation of the accumulated reward up to absorption.

Let's define ρ_i as the value returned by *function* in 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. The function *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 HOW TO INSTAL AND RUN SPNP

The package is distributed as a "tar"ed, "compress"ed, and "uuencode"d directory structure, with root directory spnp. If you receive the package by e-mail, save the message to file "spnp_distr.tar.Z.uu" and save the aditional file that you will receive as OUT and make it executable (i.e., change its mode to allow execution of the file).

You must make two decisions at this point:

- the directory where the root directory spnp will be installed on your system, let's call it Ddd;
- the platform you will use to run spnp, let's call it Ppp (you actually made that decision when you requested spnp).

For example, Ddd could be /usr/local/packages or /home/me and Ppp could be sparc or next. Then, do the following:

- move the distribution file to the correct directory

```
mv spnp_distr.tar.Z.uu Ddd
```

- go to the correct directory

```
cd Ddd
```

- execute the command

```
OUT spnp_distr
```

At this point, all the distribution files are in place. You should have:

```
Ddd/spnp/READ_ME            (a copy of this file)
Ddd/spnp/Makerun            (a makefile needed to run spnp)
Ddd/spnp/src/*.h            ("include" files needed to compile your model)
Ddd/spnp/src/testcase1.c    (a "c" file needed to check your model)
Ddd/spnp/Ppp/*.o            (compiled files forming the core of the package)
Ddd/spnp/doc                (SPNP manuals)
```

Now, define the following aliases in your ".cshrc" file (remember to **source** your ".cshrc" file if you want the aliases to become effective right away):

```
alias spnp      "make -f Ddd/spnp/Makerun DIR=Ddd PLATFORM=Ppp SPN=\!^"
alias spnpcheck "make -f Ddd/spnp/Makerun lint DIR=Ddd PLATFORM=Ppp SPN=\!^"
```

Finally, to run spnp on an input file "testcase1.c" (example supplied), you just need to type

spnp *testcase*1

Or, to check the consistency of your input file, you can type

spnpcheck *testcase*1

If inconsistencies exist between the definition of the predefined functions and their usage, they will be discovered.

Several intermediate files are generated by the package[17] 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** contaning the object module 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).

[17] Appendix D explains how to interpret the data in the intermediate files generated during the analysis

- **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.

References

[1] 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.

[2] T. Agerwala, "Putting Petri Nets to Work," *Computer*, vol. 12, no. 12, pp. 85–94, Dec. 1979.

[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 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.

[9] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] J. L. Peterson, "Petri Nets," *Computing Surveys*, vol. 9, no. 3, pp. 223–252, Sept. 1977.

[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 A: VMS Installation

The version running under the VMS operating system uses exactly the same source code, hence it has the same features as the UNIX version. 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, ...).

APPENDIX B: Complete Example of CSPL File

```
/* This is a VAX-cluster model of processor cluster system */

/* global variables: */
int    N, /* total number of processors in the cluster */
       q; /* number of processors needed for quorum    */
double c; /* coverage factor */

/* guards: */
int ensysr()  {return(mark("sysrb");}
int enrstr()  {return(mark("procup")>=q ? 1:0);}
int enprcrp() {
  return(mark("procup")>=q && mark("systr")+mark("sysbr")>0 ? 0:1);
}

/* rewards: */
reward_type availab() {return(mark("sysup"));}
reward_type ppower()  {return(mark("procup")*mark("sysup"));}

net() {

  /* places: */         /* initial markings: */
  place("procup");      init("procup",N);
  place("proctmp1");
  place("procdn");
  place("sysrb");
  place("proctmp2");
  place("sysup");       init("sysup",1);
  place("systr");

  /* transitions: */    /* input arcs: */           /* output arcs: */
  trans("trpf");        iarc("trpf","procup");      oarc("trpf","proctmp1");
                        iarc("trpf","sysup");
  trans("trpf2");       iarc("trpf2","procup");     oarc("trpf2","procdn");
  trans("trp1");        iarc("trp1","proctmp1");    oarc("trp1","systr");
                                                    oarc("trp1","procdn");
  trans("trp3");        iarc("trp3","proctmp1");    oarc("trp3","procdn");
```

```
                                                    oarc("trp3","s ysrb");
trans("trp2");        iarc("trp2","proctmp1");      oarc("trp2","procdn");
                                                    oarc("trp2","sysrb");
trans("trrs");        iarc("trrs","proctmp2");      oarc("trrs","procup");
trans("trpr");        iarc("trpr","procdn");        oarc("trpr","proctmp2");
trans("trrb");        iarc("trrb","sysrb");         oarc("trrb","sysup");
trans("trdn");        iarc("trdn","proctmp2");      oarc("trdn","systr");
                      iarc("trdn","sysup");         oarc("trdn","procup");
trans("trtr");        iarc("trtr","systr");         oarc("trtr","sysup");

/* initial marking: */
init("procup",N);
init("sysup",1);

/* inhibitor arcs: */
harc("trpf","systr");
harc("trpf2","systr");

/* timed transitions: */
ratedep("trpf",1.0/5000.0,"procup");
ratedep("trpf2",1.0/5000.0,"procup");
rateval("trpr",1.0/2.0);
rateval("trrb",6.0);
rateval("trtr",120.0);

/* firing probabilities of immediate transitions: */
probval("trp1",c);
probval("trp3",1.0);
probval("trp2",1.0-c);
probval("trrs"1.0);
probval("trdn",1.0);

/* multiple inhibitor arcs: */
mharc("trp3","procup",q);
mharc("trrs","procup",q);

/* guards: */
guard("trpf2",ensysr);
guard("trp1",enrstr);
```

```
    guard("trp2",enrstr);
    guard("trpr",enprcrp);
    guard("trrb",enrstr);
    guard("trtr",enrstr);

    /* priorities: */
    priority("trp1",10);
    priority("trp3",10);
    priority("trp2",10);
    priority("trrs",10);
    priority("trdn",10);
  }

  assert() {return(RES_NOERR);}

  ac_init() {fprintf(stderr,"\nProcessor Cluster Model\n\n");}

  ac_reach() {}

  ac_final() {
    int i;

    for(i = 1; i <= 10, i++) {
      time_value((double)i);
      pr_expected("Availability",availab);
      pr_time_avg_expected("Availability",availab);
      pr_expected("Processing Power",ppower);
      pr_cum_expected("Processing Power",ppower);
    }
  }

  parameters() {
    iopt(IOP_METHOD,VAL_TSUNIF);
    iopt(IOP_PR_FULL_MARK,VAL_YES);
    iopt(IOP_PR_MC,VAL_YES);
    iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
    iopt(IOP_PR_RGRAPH,VAL_YES);
    iopt(IOP_PR_RSET,VAL_YES);
    fopt(FOP_PRECISION,0.00000001);
```

```
    N = input("Enter the total number of processors");
    q = input("Enter number of processors needed for quorum");
    c = input("Enter the coverage factor");
  }
```

APPENDIX C: Available Options

The following is a list of available 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 transition 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: 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 should be printed together, or two separate lists should 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 is to be printed. **VAL_TANGIBLE** specifies that only the tangible markings are to 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 rate 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. 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 analyses are 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 the user to set the numerical solution method for the Markov chain, **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. **VAL_SSPOWER** specifies the power method for steady state solution that is guaranteed to converge. However, it is usually slow
- **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 iterations 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.

APPENDIX D: Format of the Intermediate Files

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

".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 transitions 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>
```

".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*).

".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 method 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.