SPNP Reference Guide Version 3.1

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

January 3, 1994

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

1 CSPL

function	**effect**
void **net**()	allows the user to completely define the structure and parameters of a SRN model
void **assert**()	allows the evaluation of a logical condition on a marking of the SRN
void **ac_init**()	used to output data about the SRN on the ''.out'' file
void **ac_reach**()	used to output data about the reachability graph on the ''.out'' file
void **ac_final**()	allows user-request outputs
void **parameters**()	allows the user to customize the package

1.1 Net function

function	**effect**
`void` **place**(*name*) `char *`*name*`;`	`defines a place with identifier` *name*
`void` **trans**(*name*) `char *`*name*`;`	`defines a transition with identifier` *name*
`void` **iarc**(*t_name*, *p_name*) `char *`*t_name*`,*`*p_name*`;`	`defines an arc directed from the place` *p_name* `to the transition` *t_name* `(input arc)`
`void` **oarc**(*t_name*, *p_name*) `char *`*t_name*`,*`*p_name*`;`	`defines an arc directed from the transition` *t_name* `to the place` *p_name* `(output arc)`
`void` **init**(*name*, *n*) `char *`*name*`;` `int` *n*`;`	`initializes the number` *n* `of tokens in any place identified by its name` *name*
`void` **harc**(*t_name*, *p_name*) `char *`*t_name*`,*`*p_name*`;`	`defines an inhibtor arc directed from the transition` *t_name* `to the place` *p_name*
`void` **miarc**(*t_name*, *p_name*, *mult*) `char *`*t_name*`,*`*p_name*`;`	`defines an input arc with multiplicity` *mult* `from the place` *p_name* `to the transition` *t_name*
`void` **moarc**(*t_name*, *p_name*, *mult*) `char *`*t_name*`,*`*p_name*`;`	`defines an output arc with multiplicity` *mult* `from the transition` *t_name* `to the place` *p_name*

Table 1: Modeling a standard Petri net

- A place/transition name is legal if:
  - its length is between 1 and **MAXNAMELENGTH**, as defined in the file, 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**}.
- By default, places are otherwise initially empty (zero tokens).

function	**effect**
void **rateval**(*name*, *val*) char **name*; rate_type *val*;	defines the rate of transition *name* as a constant value *val*
void **ratedep**(*name*, *val*, *pl*) char **name*; rate_type *val*; char **pl*;	defines the rate of transition *name* as a constant value *val* times the number of tokens in place *pl*
void **ratefun**(*name*, *func*) char **name*; rate_type (**func*)();	defines the rate of transition *name* as a general marking dependent function *func*

Table 2: Modeling timed transitions

- **rate_type** is an alias for the C type **double** (double-precision floating point number), predefined for clarity purposes only.
- You cannot disable a transition by defining a rate that evaluates to zero (0) in the marking. Instead, you must explicitly disable a transition (for example using the constructs presented in the next section).
- The package exits with an error message if a non-positive rate is found for a transition which would be otherwise enabled.

function	**effect**
void **probval**(*name*, *val*) char **name*; probability_type *val*;	defines the probability of transition *name* as a constant value *val*
void **probdep**(*name*, *val*, *pl*) char **name*; probability_type *val*; char **pl*;	defines the probability of transition *name* as a constant value *val* times the number of tokens in place *pl*
void **probfun**(*name*, *func*) char **name*; probability_type (**func*)();	defines the probability of transition *name* as a general marking dependent function *func*

Table 3: Modeling immediate transitions

- **probability_type** is an alias for the C type **double** (double-precision floating point number), predefined for clarity purposes only.

- You cannot disable a transition by defining a probability that evaluates to zero (0) in the marking. Instead, you must explicitly disable a transition (for example using the constructs presented in the next section).
- The package exits with an error message if a non-positive probability is found for a transition which would be otherwise enabled.

function	**effect**

`void` **mharc**`(`*t_name, p_name, mult*`)`
`char *`*t_name*`, *`*p_name*`;`
`int` *mult*`;`
	`defines a multiple inhibitor arc with`
	`multiplicity` *mult* `from place` *p_name*
	`to transition` *t_name*

`void` **guard**`(`*name, efunc*`)`
`char *`*name*`;`
`enabling_type (*`*efunf*`)();`
	`defines the enabling function for transition`
	name `to be` *efunc*

`void` **priority**`(`*name, prio*`)`
`char *`*name*`;`
`int` *prio*`;`
	`defines the priority for transition` *name*
	`to be` *prio*

Table 4: Modeling firing priority

- **enabling_type** is an alias for the C type **int** (integer number), meant to assume values **VAL_YES** and **VAL_NO** only.
- The function passed as actual parameter to the guards must be defined in the CSPL file before being used. It expresses marking dependency using the following predefined functions:

function	**effect**
`int` **mark**`(`*p_name*`)` `char *`*p_name*`;`	`returns the number of tokens` `in place` *p_name*
`int` **enabled**`(`*t_name*`)` `char *`*t_name*`;`	`returns 1 (`*TRUE*`) if transition` *t_name* `is enabled, 0 (`*FALSE*`) otherwise`

function	effect
void **viarc**($t\_name, p\_name, func$) char *t_name, *p_name; int (*$func$)();	defines an input arc from place p_name to transition t_name with multiplicity given by the marking dependent function $func$
void **voarc**($t\_name, p\_name, func$) char *t_name, *p_name; int (*$func$)();	defines an output arc from transition t_name to place p_name with multiplicity given by the marking dependent function $func$
void **vharc**($t\_name, p\_name, func$) char *t_name, *p_name; int (*$func$)();	defines an inhibtor arc from place p_name to transition t_name with multiplicity given by the marking dependent function $func$

Table 5: Modeling marking dependent arc multiplicity

function	effect
void **srateval**($name, val, dval$) char *$name$; rate_type $val, dval$;	defines the rate of transition $name$ and its derivative as a constant value val
void **sratedep**($name, val, dval, pl$) char *$name$; rate_type $val, dval$; char *pl;	defines the rate of transition $name$ and its derivative as a constant value val times the number of tokens in place pl
void **sratefun**($name, func, dfunc$) char *$name$; rate_type (*$func$)(),(*$dfunc$)();	defines the rate of transition $name$ and its derivative as a general marking dependent function $func$
void **sprobval**($name, val, dval$) char *$name$; probability_type $val, dval$;	defines the probability of transition $name$ and its derivative as a constant value val
void **sprobdep**($name, val, dval, pl$) char *$name$; probability_type $val, dval$; char *pl;	defines the probability of transition $name$ and its derivative as a constant value val times the number of tokens in place pl
void **sprobfun**($name, func$) char *$name$; probability_type (*$func$)(),(*$dfunc$)();	defines the probability of transition $name$ and its derivative as a general marking dependent function $func$

Table 6: Sensitivity analysis

1.2 Assert function

- The function **assert** allows the evaluation of a logical condition on a marking of the SRN.
- **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_NORERR* if the marking is (thought to be) legal.
- This check is turned off by setting the function identically egual to *RES_NOERR*.

1.3 Ac_init function

- The function **ac_init** is called before starting the reachability graph construction.
- To output data about the SRN on the ".out" file, the following function should be used:

  void pr_net_info();

1.4 Ac_reach function

- The function ac_reach is called after the reachability graph construction has completed.
- To output data about the reachability graph on the ".out" file, the following function should be used:

  void pr_rg_info();

1.5 Ac_final function

name and syntax	behavior
void **pr_mc_info**();	Output of data about the CTMC and its solution.
void **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).
void **pr_value**(*str*,*expr*) char **string*; double *expr*;	Prints the string *str* and the value of expression *expr*.
void **pr_message**(*str*) char **str*;	Allows the user to print an arbitrary message (*str*) in ".out" file.

Table 7: General functions to be used inside the ac_final function

- When either steady-state or (steady-state) sensitivity analysis is requested, the function **ac_final** is called after the solution of the **Continuous Time Markov Chain** (CTMC) has completed, to allow user-requested outputs.

name and syntax	**outputs (written on the ".out" file)**
`void` **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.
`void` **pr_std_average_der**`();`	Derivatives of all the above standard measures.

`void` **pr_expected**`(`*str*`,`*fnc*`)`
`char *`*str*`;`
`reward_type (*`*fnc*`)();`
	The string *str* and the expected value of function *fnc*.

`void` **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.

`void` **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 8: Available options for especification of output measures for steady-state analysis

- 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.

name and syntax	outputs (written on the ".out" file)
void **pr_time_avg_expected**(*fnc*) reward_type (**fnc*)();	The time-averaged expected value of function *fnc*.
void **pr_mtta**(*str*) char **str*;	The string *str* and the mean time to absorption for the SRN.
void **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.
void **cum_abs**(*fnc*) reward_type (**fnc*)();	The expected accumulated reward until absorption.

Table 9: Available options for especification of output measures for transient analysis

- For performing transient analysis and transient sensitivity analysis, a time point needs to be specified. This can be done through the function defined as:

  void time_value(t)
  double t;

- The function **pr_mtta** should be used only when the underlying CTMC has absorbing states.

- To use the function **pr_cum_abs**, the corresponding reward rate should be specified.

1.6 Parameters function

function	**effect**
`void iopt(`*option*`,`*value*`)` `int` *option*`,`*value*`;`	`enables the user to set` *option* `to have the integer` *value*
`void fopt(`*option*`,`*value*`)` `int` *option*`;` `double` *value*`;`	`enables the user to set` *option* `to have the double-precision floating point` *value*
`double input(`*msg*`)` `char *`*msg*`;`	`a message of the form ''Please type` *msg*`'' is displayed, and the user can input parameters at run-time`

Table 10: Available options for especification of output measures for transient analysis

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

- **IOP_PR_MERG_MARK** specifies whether the tangible and vanishing markings must be printed together, or two separate lists must be printed.

- **IOP_PR_FULL_MARK** specifies whether the markings are printed in long format (a full matrix indicating, for each marking, the number of tokens in each place, possibly zero), or short format (for each marking, a list of the number of tokens in the non-empty places). **VAL_YES** looks good only when the SRN has a small number of places.

- **IOP_PR_RSET** and **IOP_PR_RGRAPH** specify whether the reachability set and graph must be printed. **VAL_TANGIBLE** specifies that only the tangible markings must be printed; it cannot be used for **IOP_PR_RGRAPH**.
- **IOP_PR_MC** specifies whether the ".mc" file is generated or not.
- **IOP_PR_MC_ORDER** specifies whether the transition matrix (if **VAL_FROMTO**) or its transpose (if **VAL_TOFROM**) is printed in the ".mc" file.
- **IOP_PR_PROB** specifies whether the ".prb" file is generated or not.
- **IOP_MC** specifies the solution approach. Using **VAL_CTMC** will transform the SRN into a CTMC. Using **VAL_DTMC** will use an alternative solution approach, where the vanishing marking are not eliminated and a DTMC is instead solved. In this case, the first index in the ".mc" file is $-n$, if there are n vanishing markings, not 0. The package performs transient and sensitivity analysis by reducing the SRN to CTMC. Hence this option should be set to **VAL_CTMC** when these types of analysis is needed.
- **IOP_OK_ABSMARK**, **IOP_OK_VANLOOP**, and **IOP_OK_TRANS_M0** specify respectively whether absorbing markings, transient vanishing loops, and a transient initial marking are acceptable or not. If **VAL_NO** is specified, the program will stop if the condition is encountered. If **VAL_YES** is specified, the program will signal such occurrences, but it will continue the execution.
- **IOP_METHOD** allows to set the numerical solution method for the CTMC, **VAL_SSSOR** stands for Steady State SOR, **VAL_GASEI** stands for Steady State Gauss-Seidel, **VAL_TSUNIF** stands for Transient Solution using Uniformization. Note that there are cases where SOR does not converge, while Gauss-Seidel converges, and vice versa.
- **IOP_CUMULATIVE** specifies whether cumulative probabilities should be computed.
- **IOP_SENSITIVITY** specifies whether sensitivity analysis should be performed.
- **IOP_ITERATIONS** specifies the maximum number of iterations allowed for the numerical solution.
- **IOP_DEBUG** causes the output (on the "stderr" stream) of the markings as they are generated, and of the transitions enabled in them. It is extremely useful when debugging a SRN.
- **IOP_USENAME** specifies whether the names must be used to indicate the places and transitions involved when printing the reachability set and graph, instead of the index

(a small integer starting at 0). Using names generates a larger ".rg" file and prevents its subsequent parsing (in the current version), but it is useful when debugging a SRN.

- **FOP_ABS_RET_M0** specifies the value of the rate from each absorbing marking back to the initial marking. If this rate is positive, these markings will not correspond to absorbing states in the CTMC. This is useful to model a situation that would otherwise require a large number of transitions to model this "restart". Of course the numerical results will depend on the value specified for this option.

- **FOP_PRECISION** specifies the minimum precision required from the numerical solution. The numerical solution will stop either if the precision is reached, or if the maximum number of iteration is reached. Both the reached precision and the actual number of iterations are always output in the ".prb" file, so you can (and should) check how well the numerical algorithm performed.

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_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_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 11: Available options for the parameters function

2 Examples

2.1 Example 1

2.1.1 Source

M. K. Molloy, Performance Analysis Using Stochastic Petri Nets, *IEEE Trans. Comput.*, C-31 (9), Sept. 1982, 913–917.

2.1.2 Description

The net is shown in Figure 1

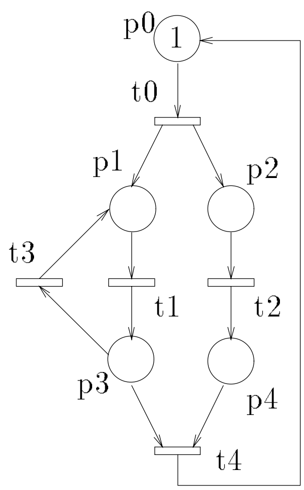


Figure 1: SRN for Example 1.

2.1.3 Features

- Assertion on place $p3$.
- Reward based functions to compute expected values.
- Default measures
- Steady-state analysis

2.1.4 SPNP File

```
/* This example adapted from M.K. Molloy's IEEE TC paper */

# include   "user.h"

/* general marking dependent reward functions: */
reward_type    ef0() { return(mark("p0")); }
reward_type    ef1() { return(mark("p1")); }
reward_type    ef2() { return(rate("t2")); }
reward_type    ef3() { return(rate("t3")); }
reward_type    eff() { return(rate("t1") * 1.8 + mark("p3") * 0.7); }

net() {

  /* places and initial markings: */
  place("p0");          init("p0",1);
  place("p1");
  place("p2");
  place("p3");
  place("p4");

  /* timed transitions and associated rates: */
  trans("t0");          rateval("t0",1.0);
  trans("t1");          rateval("t1",3.0);
  trans("t2");          rateval("t2",7.0);
  trans("t3");          rateval("t3",9.0);
  trans("t4");          rateval("t4",5.0);

  /* input arcs: */
  iarc("t0","p0");
  iarc("t1","p1");
  iarc("t2","p2");
  iarc("t3","p3");
  iarc("t4","p3");
  iarc("t4","p4");

  /* output arcs: */
  oarc("t0","p1");
  oarc("t0","p2");
  oarc("t1","p3");
  oarc("t2","p4");
  oarc("t3","p1");
  oarc("t4","p0");
}

assert() { return(mark("p3") > 5 ? RES_ERROR : RES_NOERR); }

ac_init() {
  fprintf(stderr,"\nExample from Molloy's Thesis\n\n");
  pr_net_info(); /* information on the net structure */
}
```

```
ac_reach() {
  fprintf(stderr,"\nThe reachability graph has been generated\n\n");
  pr_rg_info(); /* information on the reachability graph */
}

ac_final() {
  pr_mc_info(); /* information about the Markov chain */
  pr_expected("mark(p0)",ef0);
  pr_expected("mark(p1)",ef1);
  pr_expected("rate(t2)",ef2);
  pr_expected("rate(t3)",ef3);
  pr_expected("rate(t1) * 1.8 + mark(p3) * 0.7",eff);
  pr_std_average(); /* default measures */
}

parameters() {
  iopt(IOP_METHOD,VAL_GASEI);
  iopt(IOP_PR_FULL_MARK,VAL_YES);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_MC,VAL_YES);
  iopt(IOP_PR_PROB,VAL_YES);
  iopt(IOP_MC,VAL_CTMC);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_ITERATIONS,20000);
  fopt(FOP_PRECISION,0.00000001);
}
```

2.2 Example 2

2.2.1 Description

This example models the following piece of software:

```
A: Statements;
PARBEGIN
   B1: statements;
   B2: IF (cond1) THEN
         C: statements;
       ELSE
          DO
             D: statements;
          WHILE (cond2);
       END IF
PAREND
```

The corresponding SRN model is shown in Figure 2.

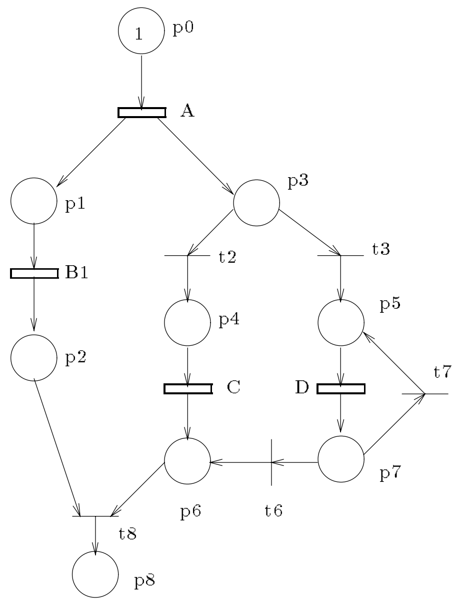


Figure 2: SRN for Example 2.

2.2.2 Features Tested

- Probability and rate functions.
- Priorities for immediate transitions.
- Reward functions.
- Transient analysis with multiple time points.

2.2.3 SPNP File

```
# include "user.h"

/*
This example corresponds to the following piece of software:

A: statements;
   PARBEGIN
   B1: statements;    B2: IF cond THEN
                          C: statements;
                          ELSE
                             DO
                             D: statements
```

```
                              WHILE cond;
                         IFEND
   PAREND
*/

/* rates and probabilities are defined as functions */
rate_type              rate0() { return(1.0); }
rate_type              rate1() { return(0.3); }
rate_type              rate4() { return(0.2); }
rate_type              rate5() { return(7.0); }
probability_type       prob2() { return(0.4); }
probability_type       prob3() { return(0.6); }
probability_type       prob6() { return(0.05);}
probability_type       prob7() { return(0.95);}
probability_type       prob8() { return(1.0); }

net() {

  /* placesand intial markings: */
  place("p0");          init("p0",1);
  place("p1");
  place("p2");
  place("p3");
  place("p4");
  place("p5");
  place("p6");
  place("p7");
  place("p8");

  /* timed transitions and associated rates: */
  trans("A");           ratefun("A", rate0);
  trans("B1");          ratefun("B1",rate1);
  trans("C");           ratefun("C", rate4);
  trans("D");           ratefun("D", rate5);

  /* immediate transitions and associated priorities: */
  trans("t2");          probfun("t2",prob2);          priority("t2",1);
  trans("t3");          probfun("t3",prob3);          priority("t3",1);
  trans("t6");          probfun("t6",prob6);          priority("t6",1);
  trans("t7");          probfun("t7",prob7);          priority("t7",1);
  trans("t8");          probfun("t8",prob8);          priority("t8",1);

  /* input arcs: */
  iarc("A","p0");
  iarc("B1","p1");
  iarc("t2","p3");
  iarc("t3","p3");
  iarc("C","p4");
  iarc("D","p5");
  iarc("t6","p7");
  iarc("t7","p7");
  iarc("t8","p2");
  iarc("t8","p6");
```

```
  /* output arcs: */
  oarc("A","p1");
  oarc("A","p3");
  oarc("B1","p2");
  oarc("t2","p4");
  oarc("t3","p5");
  oarc("C","p6");
  oarc("D","p7");
  oarc("t6","p6");
  oarc("t7","p5");
  oarc("t8","p8");
}

assert() { return(RES_NOERR); }

ac_init() { fprintf(stderr,"\nSoftware modeling example\n\n"); }

ac_reach() {}

reward_type rfunc() { return(mark("p8")); }

ac_final() {
  int i;

  /* Transient analysis with multiple time points */
  /* reward function */
  for(i = 1; i < 10; i++) {
    time_value( (double) i );
    pr_expected("probability of completion", rfunc);
  }
  for(i = 10; i <= 20; i += 2) {
    time_value( (double) i );
    pr_expected("probability of completion", rfunc);
  }
  for(i = 20; i <= 50; i += 5) {
    time_value( (double) i );
    pr_expected("probability of completion", rfunc);
  }
}

parameters() { iopt(IOP_METHOD,VAL_TSUNIF); /* Transient analysis */ }
```

2.3 Example 3

2.3.1 Description

This example models a finite-buffer $M/M/m/b$ queue shown in Figure 3. The corresponding SRN is shown in Figure 4.

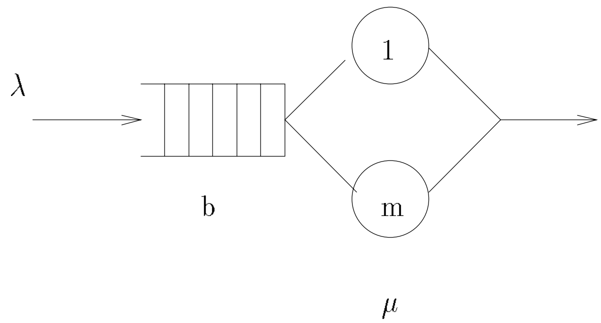
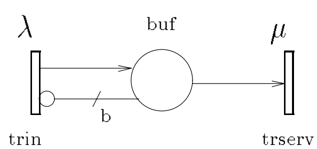


Figure 3: The $M/M/m/b$ Queue.



Transition	Rate Function
trserv	$\#(buf)\mu$ if $(\#(buf) < m)$ $m\mu$ otherwise

Figure 4: SRN for Example 3.

2.3.2 Features

- Both steady-state and transient analysis.
- Marking dependent firing rates.
- Assertions.
- General reward specification.

2.3.3 SPNP File

```
/* This example models a Multi-server FCFS queue with finite buffer */
/* An M/M/m/b queue */

# include   "user.h"

/* global variables: */

int b,           /* number of buffers */
```

```
    m,          /* number of servers */
    method;     /* method of analysis */

double lambda,
       mu;

/* marking dependent firing rate */
rate_type     rate_serv()     { return(mark("buf") < m) ? mark("buf")*mu : m*mu); }

/* rewards: */
reward_type   qlength()       { return(mark("buf")); }
reward_type   util()          { return(enabled("trserv")); }
reward_type   tput()          { return(rate("trserv")); }
reward_type   probrej()       { return(mark("buf") == b ? 1.0 : 0.0); }
reward_type   probempty()     { return(mark("buf") == 0 ? 1.0 : 0.0); }
reward_type   probhalffull()  { return(mark("buf") == b/2 ? 1.0 : 0.0); }

net() {

  /* places: */
  place("buf");

  /* timed transitions and associated rates: */
  trans("trin");        rateval("trin",lambda);
  trans("trserv");      ratefun("trserv",rate_serv);

  /* input arcs: */
  iarc("trserv","buf");

  /* output arcs: */
  oarc("trin","buf");

  /* inhibtor arcs: */
  mharc("trin","buf",b);
}

assert() {
  /* Make sure that the number of tokens in buf does not exceed the  buffer size */
  if( mark("buf") > b )
    return(RES_ERROR);
  else
    return(RES_NOERR);
}

ac_init() {
  fprintf(stderr,"A model of the M/M/m/b Queue");
  pr_net_info();
}

ac_reach() { pr_rg_info(); }

ac_final() {
  double time_pt;
```

```
  /* measures related to the queue */
  if( method == 0 ) {
    pr_expected("Average Queue Length", qlength);
    pr_expected("Average Throughput", tput);
    pr_expected("Utilization", util);
    /* this case corresponds to buf having b tokens */
    pr_expected("Probability of rejection",probrej);
    /* this case corresponds to buf having zero tokens */
    pr_expected("Probability that queue is empty",probempty);
    /* this case corresponds to buf having b/2 tokens */
    pr_expected("Probability that queue is half full",probhalffull);
  } else {
    for( time_pt = 0.1; time_pt < 1.0; time_pt += 0.1 ) {
      time_value(time_pt);
      pr_expected("Average Queue Length", qlength);
      pr_expected("Average Throughput", tput);
      pr_expected("Utilization", util);
      /* this case corresponds to buf having b tokens */
      pr_expected("Probability of rejection",probrej);
      /* this case corresponds to buf having zero tokens */
      pr_expected("Probability that queue is empty",probempty);
      /* this case corresponds to buf having b/2 tokens */
      pr_expected("Probability that queue is half full",probhalffull);
    }
    for( time_pt = 1.0; time_pt < 10.0; time_pt += 1.0 ) {
      time_value(time_pt);
      pr_expected("Average Queue Length", qlength);
      pr_expected("Average Throughput", tput);
      pr_expected("Utilization", util);
      /* this case corresponds to buf having b tokens */
      pr_expected("Probability of rejection",probrej);
      /* this case corresponds to buf having zero tokens */
      pr_expected("Probability that queue is empty",probempty);
      /* this case corresponds to buf having b/2 tokens */
      pr_expected("Probability that queue is half full",probhalffull);
    }
  }
}

parameters() {
  method = input("Input 0/1 for Steady-state/Transient analysis");
  if( method == 0 )
    iopt(IOP_METHOD,VAL_SSSOR);
  else if( method == 1 )
    iopt(IOP_METHOD,VAL_TSUNIF);
  else {
    fprintf(stderr,"ERROR: Illegal method specification");
    exit(1);
  }
  iopt(IOP_PR_FULL_MARK,VAL_YES);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_MC,VAL_YES);
```

```
  iopt(IOP_PR_PROB,VAL_YES);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_ITERATIONS,20000);
  iopt(IOP_CUMULATIVE,VAL_NO);
  fopt(FOP_PRECISION,0.00000001);
  lambda = input("Enter lambda");
  mu = input("Enter mu");
  b = input("Enter the number of buffers");
  m = input("Enter the number of servers");
}
```

2.4 Example 4

2.4.1 Source

J. T. Blake, A. L. Reibman and K. S. Trivedi, Sensitivity Analysis of Reliability and Performability Measures for Multiprocessor Systems, *Proc. 1988 ACM SIGMETRICS*, Santa Fe, NM, 1988.

2.4.2 Description

This example models the C.mmp system designed at CMU. The architecture of the system is shown in Figure 5. The corresponding SRN model is shown in Figure 6.

2.4.3 Features

- Enabling functions.
- Variable multiplicity arcs.
- Reward based measures.
- Transient analysis.

2.4.4 SPNP File

```
/*
This is a model of the C.MMP multiprocessor system adopted from Blake,
Reibman and Trivedi "Sensitivity Analysis of Reliability and
Performability Measures for Multiprocessor Systems", ACM SIGMETRICS 1988.
*/

# include <math.h>
# include "user.h"

# define min(x,y) ((x < y) ? x : y )
```

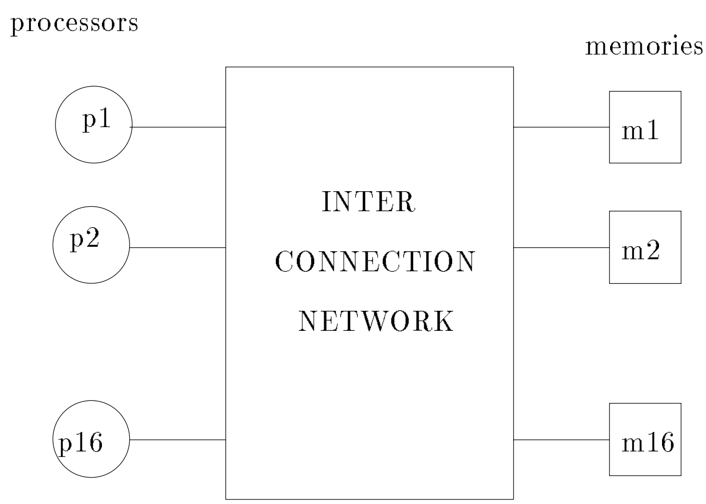
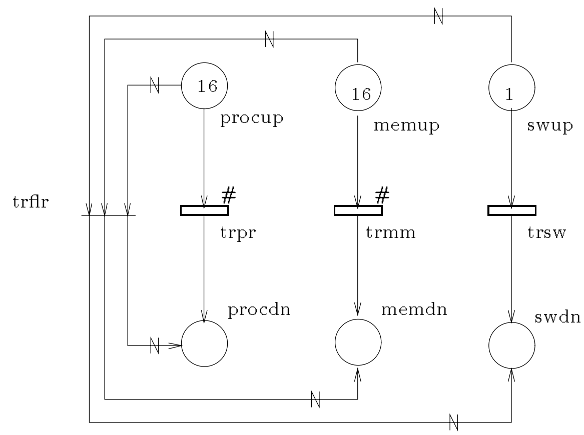


Figure 5: The C.mmp Architecture.

```
# define max(x,y) ((x > y) ? x : y )

extern int abs();
extern double pow();

int k;

int apfl()  { return( mark("procup")); }
int amfl() { return( mark("memup")); }
int asfl()  { return( mark("swup")); }

/* enabling functions: */

enabling_type entrflr() {
  if( mark("procup") == 0 && mark("memup") == 0 && mark("swup") == 0 )
    return(0);
  if( mark("procup") < k || mark("memup") < k || mark("swup") == 0 )
    return(1);
  else
    return(0);
}

/* rewards: */

reward_type reliab()        {
  return(mark("procup") ≥ k && mark("memup") ≥ k && mark("swup") == 1 ? 1.0 : 0.0);
}
reward_type reward_rate() {
```



Transition	Enabling Function
trflr	$((\#(procup) < k) \vee (\#(memup) < k) \vee (\#(swup) = 0))$ $\wedge((\#(procup) > 0) \vee (\#(memup) > 0) \vee (\#(swup) > 0))$

Arcs	Multiplicity Function
procup → trflr & trflr → procdn	$\#(procup)$
memup → trflr & trflr → memdn	$\#(memup)$
swup → trflr & trflr → swdn	$\#(swup)$

Figure 6: SRN for Example 4.

```
  double m, l, temp;

  if( mark("procup") ≥ k && mark("memup") ≥ k && mark("swup") == 1 ) {
    l = min((double)mark("procup"),(double)mark("memup"));
    m = max((double)mark("procup"),(double)mark("memup"));
    temp = pow( (1.0 − (1.0 / m)) , l );
    return( m ∗ (1.0 − temp) );
  } else
    return(0);
}

net() {

  /* places and initial markings: */
  place("procup");       init("procup",16);
  place("procdn");
  place("memup");        init("memup",16);
  place("memdn");
  place("swup");         init("swup",1);
```

```
  place("swdn");

  /* timed transitions: */
  trans("trpr");      ratedep("trpr",0.0000689,"procup");
  trans("trmm");      ratedep("trmm",0.000224,"memup");
  trans("trsw");      rateval("trsw",0.0002202);

  /* immediate transition and associated probability, priority and guard: */
  trans("trflr");     probval("trflr",1.0);     priority("trflr",100);     guard("trflr",entrflr);

  /* input arcs: */
  iarc("trpr","procup");
  iarc("trmm","memup");
  iarc("trsw","swup");

  /* output arcs: */
  oarc("trpr","procdn");
  oarc("trmm","memdn");
  oarc("trsw","swdn");

  /* multiple input arcs: */
  viarc("trflr","procup",apfl);
  viarc("trflr","memup",amfl);
  viarc("trflr","swup",asfl);

  /* multiple output arcs: */
  voarc("trflr","procdn",apfl);
  voarc("trflr","memdn",amfl);
  voarc("trflr","swdn",asfl);
}

assert() {}

ac_init() {
  fprintf(stderr,"\nC.MMP Reliability Model\n\n");
  pr_net_info();
}

ac_reach() { pr_rg_info(); }

ac_final() {
  double time_pt;

  for ( time_pt = 500.0; time_pt < 5000.0; time_pt+= 500.0 ) {
    time_value( time_pt );
    pr_expected("Reliability",reliab);
    pr_expected("Expected Reward",reward_rate);
    pr_cum_expected("Expected Accumulated Reward",reward_rate);
  }
}

parameters() {
  iopt(IOP_METHOD,VAL_TSUNIF);
```

```
  iopt(IOP_PR_FULL_MARK,VAL_YES);
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  fopt(FOP_PRECISION,0.00000001);
  k = input("Please input min. number of proc. and mem. needed");
  if( k < 1 ) {
    fprintf(stderr,"ERROR: atleast one processor is needed (k > 1)");
    exit(1);
  }
}
```

3 Example 5

3.0.5 Source

P. Hiedelberger and A. Goyal, Sensitivity Analysis of Continuous Time Markov chains using Uniformization, *Computer Performance and Reliability*, G. Iazeolla, P. J. Courtois and O. J. Boxma (Eds.), Elsevier Science Publishers, B.V. (North-Holland), Amsterdam, 1988.

3.0.6 Description

This example is a model of a database system shown in Figure 7.

The system consists of a front end (FE), a database (DB) and two processing sub-systems. Each processing sub-system consists of two processors (P), a memory (M) and a switch (S). For the system to be functional, we need at least one of the processing sub-systems to be operational. The database and the front-end should also be operational. The processing sub-system is functional as long as the memory, the switch and at least one of the processors is functional. When a processor fails, with probability c it fails without disturbing the system. However, with probability $1 - c$ the failing processor corrupts the database causing it to fail and consequently rendering the system unoperational. The processors, memories and switches can be repaired while the system is up. The memories and switches receive priority over the processors for repair. The corresponding SRN model is shown in Figure 8.

3.0.7 Features

- Global variables.
- Enabling function.
- Reward based functions.
- Transient analysis.

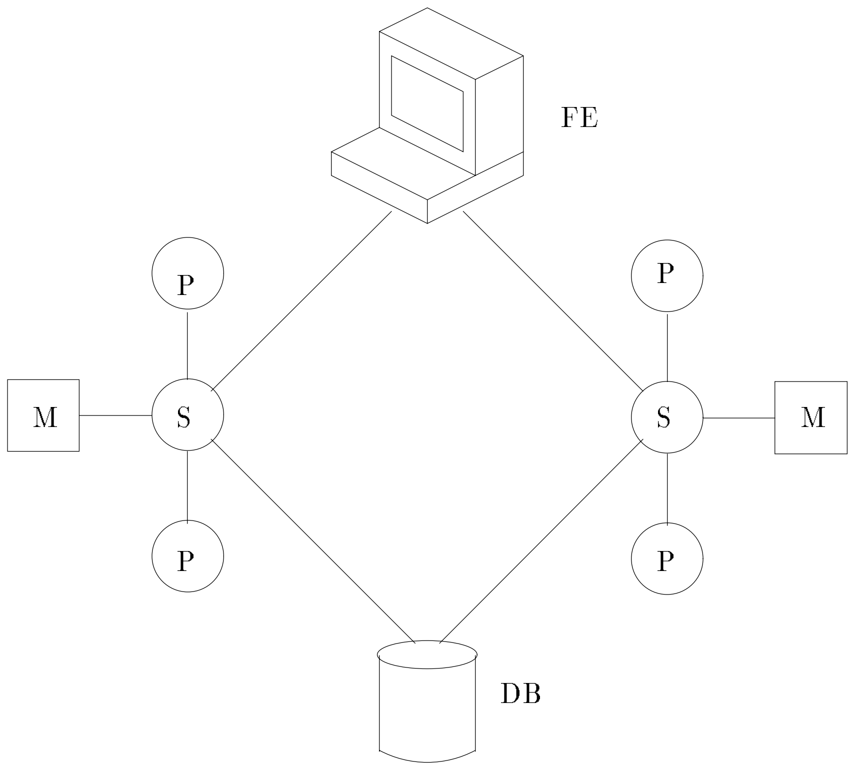


Figure 7: The Database System Architecture.

3.0.8 SPNP File

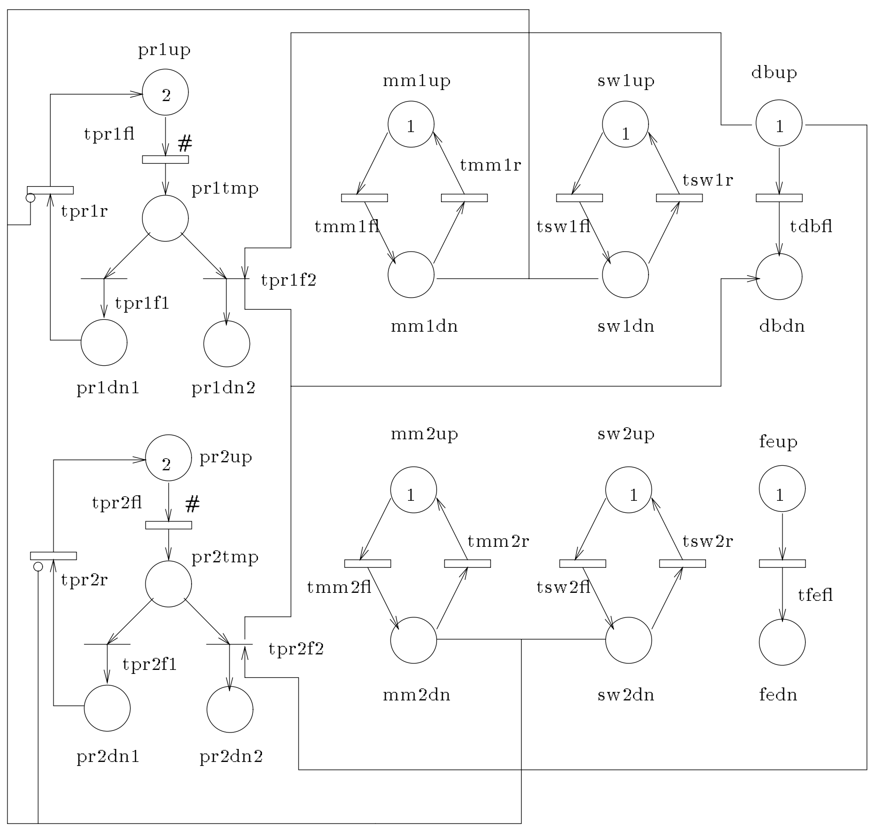
```
/* This is a petri-net model of the database system example from
the paper on sensitivity by Hiedelberger and Goyal */

# include   "user.h"

int count = 0;
double coverage = 0.99;

/* enabling functions: */

enabling_type enall() {
  /* if the database is failed */
  if( mark("dbup") == 0 )
    return(0);
  /* if the front end is failed */
  if( mark("feup") == 0 )
    return(0);
```

```
  /* if both the processing sub-systems are failed */
  if(( mark("mm1up") == 0 || mark("sw1up") == 0 || mark("pr1up") == 0 ) &&
     ( mark("mm2up") == 0 || mark("sw2up") == 0 || mark("pr2up") == 0 ))
    return(0);

  return(1);
}

/* rewards: */

reward_type reliab() {
  /* if the database is failed */
  if( mark("dbup") == 0 )
    return(0.0);
  /* if the front end is failed */
  if( mark("feup") == 0 )
    return(0.0);
  /* if both the processing sub-systems are failed */
  if(( mark("mm1up") == 0 || mark("sw1up") == 0 || mark("pr1up") == 0 ) &&
     ( mark("mm2up") == 0 || mark("sw2up") == 0 || mark("pr2up") == 0 ))
    return(0.0);
  return(1.0);
}

net() {

  /* places and initial markings of the first processing system: */
  place("mm1up");        init("mm1up",1);
  place("sw1up");        init("sw1up",1);
  place("pr1up");        init("pr1up",2);
  place("mm1dn");
  place("sw1dn");
  place("pr1tmp");
  place("pr1dn1");
  place("pr1dn2");

  /* places and initial markings of the second processing system: */
  place("mm2up");        init("mm2up",1);
  place("sw2up");        init("sw2up",1);
  place("pr2up");        init("pr2up",2);
  place("mm2dn");
  place("sw2dn");
  place("pr2tmp");
  place("pr2dn1");
  place("pr2dn2");

  /* places and initial markings of the database processor: */
  place("dbup");         init("dbup",1);
  place("dbdn");

  /* places and initial markings of the front end processor: */
  place("feup");         init("feup",1);
  place("fedn");
```

```
/* timed transitions, respective rates and guards: */
trans("tmm1fl");   rateval("tmm1fl",1000./2400.);         guard("tmm1fl",enall);
trans("tsw1fl");   rateval("tsw1fl",1000./2400.);         guard("tsw1fl",enall);
trans("tpr1fl");   ratedep("tpr1fl",1000./2400.,"pr1up"); guard("tpr1fl",enall);
trans("tmm1r");    rateval("tmm1r",1000.);                guard("tmm1r",enall);
trans("tsw1r");    rateval("tsw1r",1000.);                guard("tsw1r",enall);
trans("tpr1r");    rateval("tpr1r",1000.);                guard("tpr1r",enall);
trans("tmm2fl");   rateval("tmm2fl",1000./2400.);         guard("tmm2fl",enall);
trans("tsw2fl");   rateval("tsw2fl",1000./2400.);         guard("tsw2fl",enall);
trans("tpr2fl");   ratedep("tpr2fl",1000./2400.,"pr2up"); guard("tpr2fl",enall);
trans("tmm2r");    rateval("tmm2r",1000.);                guard("tmm2r",enall);
trans("tsw2r");    rateval("tsw2r",1000.);                guard("tsw2r",enall);
trans("tpr2r");    rateval("tpr2r",1000.);                guard("tpr2r",enall);
trans("tdbfl");    rateval("tdbfl",1000./2400.);          guard("tdbfl",enall);
trans("tfefl");    rateval("tfefl",1000./2400.);          guard("tfefl",enall);

/* immediate transitions, respective probabilities and priorities: */
trans("tpr1f1");   probval("tpr1f1",coverage);            priority("tpr1f1",100);
trans("tpr1f2");   probval("tpr1f2",1.0 − coverage);      priority("tpr1f2",100);
trans("tpr2f1");   probval("tpr2f1",coverage);            priority("tpr2f1",100);
trans("tpr2f2");   probval("tpr2f2",1.0 − coverage);      priority("tpr2f2",100);

/* input arcs: */
iarc("tmm1fl","mm1up");
iarc("tsw1fl","sw1up");
iarc("tpr1fl","pr1up");
iarc("tpr1f1","pr1tmp");
iarc("tpr1f2","pr1tmp");
iarc("tpr1f2","dbup");
iarc("tmm1r","mm1dn");
iarc("tsw1r","sw1dn");
iarc("tpr1r","pr1dn1");
iarc("tmm2fl","mm2up");
iarc("tsw2fl","sw2up");
iarc("tpr2fl","pr2up");
iarc("tpr2f1","pr2tmp");
iarc("tpr2f2","pr2tmp");
iarc("tpr2f2","dbup");
iarc("tmm2r","mm2dn");
iarc("tsw2r","sw2dn");
iarc("tpr2r","pr2dn1");
iarc("tdbfl","dbup");
iarc("tfefl","feup");


/* output arcs: */
oarc("tsw1fl","sw1dn");
oarc("tpr1fl","pr1tmp");
oarc("tpr1f1","pr1dn1");
oarc("tpr1f2","pr1dn2");
oarc("tpr1f2","dbdn");
```

```
  oarc("tmm1r","mm1up");
  oarc("tsw1r","sw1up");
  oarc("tpr1r","pr1up");
  oarc("tmm2fl","mm2dn");
  oarc("tsw2fl","sw2dn");
  oarc("tpr2fl","pr2tmp");
  oarc("tpr2f1","pr2dn1");
  oarc("tpr2f2","pr2dn2");
  oarc("tpr2f2","dbdn");
  oarc("tmm2r","mm2up");
  oarc("tsw2r","sw2up");
  oarc("tpr2r","pr2up");
  oarc("tdbfl","dbdn");
  oarc("tfefl","fedn");


  /* inhibtor arcs: */
  harc("tpr1r","mm1dn");
  harc("tpr1r","mm2dn");
  harc("tpr1r","sw1dn");
  harc("tpr1r","sw2dn");
  harc("tpr2r","mm1dn");
  harc("tpr2r","mm2dn");
  harc("tpr2r","sw1dn");
  harc("tpr2r","sw2dn");
}

assert() {
  /* count the number of states in which the failure of the database
     by itself has caused system failure. This excludes the states
     in which the database has been corrupted by a failing processor
  */
  if( mark("dbdn") == 1 && mark("pr1dn2") == 0 && mark("pr2dn2") == 0 )
    count++;
  return(RES_NOERR);
}

ac_init() { fprintf(stderr,"\nExample from Heidelberger & Goyal\n\n"); }

ac_reach() {}

ac_final() {
  double time_pt;

  for( time_pt = 0.1; time_pt <= 1.0; time_pt+= 0.1 ) {
    time_value(time_pt );
    pr_expected("Reliability:",reliab);
  }
  pr_value("No. of States in which DB caused failure",(double)count);
}

parameters() {
  iopt(IOP_METHOD,VAL_TSUNIF);
```

```
  iopt(IOP_PR_MC_ORDER,VAL_TOFROM);
  iopt(IOP_PR_RSET,VAL_YES);
  iopt(IOP_PR_RGRAPH,VAL_YES);
  iopt(IOP_CUMULATIVE,VAL_NO);
  fopt(FOP_PRECISION,0.00000001);
}
```



Transition	Enabling Function
all	$(\#(dbup) = 1) \wedge (\#(feup) = 1)$ $\wedge((\#(pr1up) > 0) \wedge (\#(mm1up) > 0) \wedge (\#(sw1up) > 0)$ $\vee(\#(pr2up) > 0) \wedge (\#(mm2up) > 0) \wedge (\#(sw2up) > 0))$

Figure 8: SRN for Example 5.