SHARPE: Symbolic Hierarchical Automated Reliability/Performance Evaluator
Language Description

Robin Sahner and K.S. Trivedi

Contents

1 Introduction

SHARPE (Symbolic Hierarchical Automated Reliability/Performance Evaluator) is a program that accepts specifications of mathematical models and requests for model analysis. It supports the following model types:

- Markov chains(acyclic, irreducible and phase type)
- Semi-Markov chains (acyclic and irreducible)
- Reliability block diagrams
- Fault trees
- Reliability graphs
- Single-chain product form queuing networks
- Multiple-chain product form queuing networks
- Generalized stochastic Petri nets
- Series-parallel acyclic graphs

This document is a reference manual for the SHARPE specification language. The sections contain the following information:

Section 2: Comments, constants, names, words, evaluated words, expressions, variables, functions, and lists of parameters and arguments.

Section 3: Specifications of distributions

Section 4: Specifications of models

Section 5: Built-in functions

Section 6: Asking for results

Section 7: Controlling the analysis process

Section 8: Input size limitations

Section 9: Summary of top-level input statements

Section 10: Examples

We use the following conventions:

- Keywords and necessary punctuation are given in boldface.
- Syntactic categories are given in italics.
- A line contained in angled brackets < > indicates and unspecified number (possibly zero) of repetitions of the line.
- Curly brackets {} indicate an optional portion of a line or optional set of lines.

- Square brackets [] containing elements separated by vertical lines ("|") indicates that one of the elements is expected.

SHARPE distinguishes between lower-case and upper-case letters. Keywords must be either all lower-case or all upper-case. In this guide, keywords are shown in lower-case.

The SHARPE language is line-oriented. Tokens within a line may be separated by any amount of white space (defined to be blanks and/or tabs). SHARPE recognizes the UNIX line-continuation character, backslash ("\").

2 Basic language components

2.1 Comments

Any line that has the character "*" (star) as its first now-white space character is considered to be a comment line, and is ignored.

2.2 Copying text to output

A line of the form:

> **echo** *anytext*

causes *anytext* to be written on the output and is otherwise ignored.

2.3 Constants

A *constant* is an integer (sequence of digits) or a real number (digits followed by a decimal point followed by digits). In a real number, either the leading or trailing digits may be omitted (but not both). All integers are converted to floating-point format for purposes of internal computation. SHARPE does not currently support scientific notation for constant input (for example SHARPE does not understand the meaning of 3.1E+6)

2.4 Names

A *name* consists of a letter followed by any number of letters and/or digits. *Names* are used for variables, function and distribution names, parameters, and indices in **sum** functions. *Names* may be any length, but SHARPE only looks at the first 14 characters (if the lengths of names used is larger than 14 characters, it would be a good idea to keep some characters in the first 14 characters different).

2.5 Words and Evaluated Words

A *word* is a sequence of any characters except white space, commas, semicolons, parentheses, backslashes and dollar signs. Words are used for the specification of names of models and the components they contain.

A *subword* is a string of the form $\$n$ or $\$(expression)$. In the first case, n is any single letter. In the second case, any expression can be used within the parentheses. An *evaluated word* is made up of *subwords* and any characters except white space, commas, semicolons, parentheses, backslashes and dollar signs.

When component names are used within built-in functions, they are *evaluated words*. The use of *evaluated words* provides a limited means of indexing. If i is 4 and j is 5, the *evaluated word* $\$(i)$A-$\$(j-i)$B-$\$j$ evaluates to the component names 4A-1B-5. For examples of the use of *evaluated words*, see sections 10.1 and 10.2.

Words may be any length, but SHARPE only looks at first 14 characters. In the case of *evaluated words*, the truncation occurs after evaluation. In the remainder of this document, identifiers ending in *_name* are assumed to be of type *name*, identifiers ending in *_word* are assumed to be of type *word*, and identifiers ending in *_eword* are assumed to be of type *evaluated word.*

2.6 Arithmetic Expressions

Expressions (*expression*)are written in infix form. The following operands are allowed:

+ addition

- subtraction (binary) or negation (unary)

/ division (floating-point)

$*$ multiplication

^ exponentiation (binary) or "power of e" (unary)

The use of a unary "^" to mean "power of e" is nonstandard. At first, it seemed natural to define a built-in variable name **e**, but if that is done, we are prevented from allowing the letter **e** as a user-defined variable.

The default operator precedence is as follows:

1. negation
2. exponentiation and "power of e"
3. multiplication and division
4. addition and subtraction

Within each level, evaluation is done from left to right. An order of evaluation other than the default may be forced by the use of parentheses. The allowed operands are as follows:

- *constant*
- *simple_var*
- *defined_var*
- *func_name* (*arg_list*)
- *built_in_function*

If *func_name* is used, it must have already have been defined and the number of arguments must agree with the number of parameters (possibly none) specified when the function was defined.

Built-in functions are described in section 5. If a built-in function containing a model argument is used, the model argument must have been defined. The number of arguments (not counting the system and node names) must match the number of parameters (possibly none) in the system definition.

Examples of expressions are:

```
360 / (360 + (k - 1) * x)
k * x * (1 - c(x,k))
3 * lambda * prob (second-fault, recovered; 2*lambda)
^(-(k-1) * x * tau)
delta / ((k-1) * x) * (1 - ^(-(k-1) * x / delta))
sum (i,1,n,sum(j,1,i,j))
sum (i,1,n,prob(m,$(i)-$(j)))
```

A *constant_expression* is an expression in which all the operands are *constants*.

2.7 Variables, Binding and Functions

A simple variable (*simple_var*) is a *name* that is defined implicitly by appearing in an expression. All simple variables must be bound to values before analysis takes place. There are two formats for binding variables. A single variable can be bound to a value by the line:

bind *simple_var expression*

A group of variables can be bound as follows:

bind
<*simple_var expression*>
end

Simple variables can be bound either before or after their first appearance in an expression. All variables appearing in the specification of a model must be bound to values before the user requests results from that model.

Once a variable is bound, it retains its value until it is bound to a different expression.

A *function* is defined by:

> **function** *func_name* (*param_list*) *expression*

The parameter list is allowed to be empty, but the parentheses must be present.

A defined variable (*defined_var*) is the same as a function with no arguments; it is defined by:

> **var** *defined_var expression*

If a function or defined variable contains a simple variable in its definition and the simple variable is rebound, the function or defined variable is recomputed the next time its value is needed.

2.8 Scope of Names and Words

Simple variable, defined variables, function names, distribution names and model names are global. They must all be different. No two models can have the same name, even if they are of different type.

Component names are local to the model in which they appear.

Parameter names are local to their system, function, or distribution definition. The index of a loop is local to the loop. The index in a sum function is local to the function.

2.9 Parameter and Argument Lists

A parameter list (*param_list*) has the form:

$$name, name, \cdots, name$$

Parameter lists are used when in the definition of functions, distributions and models.

An argument list (*arg_list*) has the form

$$expression, expression, \cdots, expression$$

Argument lists are used when functions or models are to be evaluated. It is possible for a parameter or argument list to be empty.

3 Specifications of distributions

A cumulative distribution function (CDF) is a finite exponential polynomial:

$$F(t) = \sum_j a_j t^{k_j} e^{b_j t} \qquad (t \geq 0)$$

There are seven built-in forms for specifying a distribution (*dist*):

zero This specifies the discrete distribution having all its mass at zero. ($F(t) = 1,\ t \geq 0$)

inf This specifies the discrete distribution having all its mass at infinity. ($F(t) = 0,\ t \geq 0$)

gen *triple*, *triple*, $\cdots$

This specifies a complete exponential polynomial, term by term, where all numbers are real (not complex). Each triple has the form:

a_j, k_j, b_j

where a_j and b_j may be real or integer, and k_j must be a non-negative integer. b_j must be less than zero.

cgen *five-tuple, five-tuple,* $\cdots$

This specifies a complete exponential polynomial, term by term, where complex numbers are used. Each *five-tuple* has the form

$real(a_j), imag(a_j), k_j, real(b_j), imag(b_j)$

where k_j must be non-negative integer, and the rest of the numbers may be real or integer. $real(b_j)$ must be less than zero. The distribution thus specified must be real valued. an equivalent condition is that the imaginary numbers must occur in conjugate pairs. That is, for every appearance of the term $real(a_j), imag(a_j), k_j, real(b_j), imag(b_j)$ where $imag(a_j)$ or $imag(b_j)$ is nonzero, the term $real(a_j), -imag(a_j), k_j, real(b_j), -imag(b_j)$ must appear.

tgen *n-tuple, n-tuple,* $\cdots$

Here *n* is either four or five for each term. This specifies a complete exponential polynomial using sine and cosine functions. Each *n-tuple* has one of the following three forms:

$a_j, k_j, b_j,$ **none**
$a_j, k_j, b_j,$ **sin**,x_j
$a_j, k_j, b_j,$ **cos**,x_j

where k_j must be a non-negative integer and the rest of the numbers may be real or integer. b_j must be less than zero.

defined_dist (arg_list)

> This specifies a built-in or user-defined distribution. The parentheses must be present even if there are no parameters. There is one built-in distribution, **exp**, having one parameter that is the inverse of the mean of the distribution. If the parameter is λ, then $F(t) = 1 - e^{-\lambda t}$.

cdf (*system_name* {,*state_eword*} {;*arg_list*})

> The *system_name* may be that of any model type except irreducible chain or gspn. A *state_eword* is allowed only for Markov and semi-Markov chains (with absorbing states). The distribution is that printed by the **cdf** keyword (see (1) and (2) in section 6.5). The number of arguments must match the number of parameters (possibly none) in the system definition.
>
> To create a user-defined distribution, the following statement is used:

poly *name* (*param_list*) *dist*

> The parentheses must be present even if the argument list is empty. *name* becomes a defined distribution (*defined_dist*) and can be used anywhere one of the built-in forms can be used.

4 Specification of models

A model may have parameters; in that case the scope of the parameters is exactly the entire model definition. If there are no parameters, the parentheses on the first line on the model specification may be (but do not have to be) left out.

The comments in the following model specifications are not required; they are included here for informational value.

4.1 Markov chains

SHARPE allows three kinds of Markov chains; irreducible, acyclic and PH-type. A PH-type (phase type) chain is a chain with absorbing states in which every state that is not absorbing is transient.

An irreducible Markov chain (discrete-time or continuous-time) is specified in one of the two ways, either with or without initial state probabilities. Only for the irreducible case, SHARPE solves both continuous and discrete time irreducible Markov chains. A discrete time Markov chain is defined as if it were a continuous time Markov chain. We emphasize that SHARPE is not designed to solve acyclic or phase-type discrete-time Markov chains or is it designed to carry out the transient analysis of discrete-time Markov chains. In these case, SHARPE handles continuous-time Markov chains.

While defining a discrete time Markov chain, eliminate all the self loops and specify transition probabilities in place of transition rates (defined for continuous time Markov chains).

4.1.1 Markov chains with absorbing states

A (continuous-time) markov chain with absorbing states (either acyclic or PH-type) is specified as follows:

> **markov** *name* {*(param_list)*}
> $*$ section 1: transitions and transition rates
> <*name name expression*>
> $*$ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**
> $*$ section 3: initial state probabilities
> <*name expression*>
> **end**

Each line in the first section specifies a state transition from the first *name* to the second *name* having as its transition rate the given *expression*. The state transitions (and associated rates) can be given in any order.

The second section is optional. If present, each line assigns a reward rate (*expression*) to a non-absorbing state (*name*). In the current implementation, non-absorbing states must have non-zero reward rates. By default, a state that is not assigned a reward rate is assumed to have a reward rate of zero. If the **reward** keyword is followed by **default** *expression*, the default reward for this chain is changed to the *expression*.

The third section gives initial state probabilities. In each line, the node *name* is assigned *expression* as its initial state probability. If a state is not assigned an initial state probability, the probability is assumed to be zero. The sum of all assigned initial probabilities must be one.

For an acyclic chain, the third section may be left empty if there is a single node having no incoming transitions. In that case, the single node is assumed to have an initial probability of one, and all other nodes have initial probability zero. If more than one node has no incoming probability, this section must not be empty.

4.1.2 Irreducible continuous time markov chains

If only a steady-state analysis of an irreducible continuous-time Markov chain is to be done (using the built-in function **prob**), initial state probabilities are irrelevant. If a transient analysis is to be done (using **tvalue**), initial state probabilities are required.

Older versions of SHARPE did not support transient analysis of continuous-time irreducible chains and did not expect initial state probabilities to be specified for irreducible chains. For the sake of compatibility, SHARPE assumes that continuous-time irreducible chains will be followed by initial state probabilities if and only if the line containing the chain name ends in the keyword **readprobs**. A continuous-time irreducible chain without initial state probabilities is specified as follows:

> **markov** *name* {*(param_list)*}
> ∗ section 1: transitions and transition rates
> <*name name expression*>
> ∗ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**

Once a continuous-time irreducible chain has been specified without initial state probabilities, **tvalue** cannot be applied to it. A continuous-time irreducible chain with initial state probabilities is specified as follows:

> **markov** *name* {*(param_list)*} **readprobs**
> ∗ section 1: transitions and transition rates
> <*name name expression*>
> ∗ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**
> ∗ section 3: initial state probabilities
> <*name expression*>
> **end**

When a continuous-time irreducible chain has been specified this way, either **prob** or **tvalue** can be applied to it.

4.1.3 Irreducible discrete time markov chains

For doing the steady state analysis of a discrete-time Markov chain (using **prob**), initial probabilities are not required. Note that a transient analysis of such chains cannot be carried out in SHARPE. A discrete-time Markov chain without initial state probabilities is specified as follows:

> **markov** *name* {*(param_list)*}
> ∗ section 1: transitions and transition probabilities
> <*name name expression*>
> ∗ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**

Each line in the first section specifies a state transition from the first *name* to the second *name* having as its transition probability the given *expression.* Alternatively the transition probabilities may be specified directly. The sum of the transition probabilities for any *name* may not equal one due to elimination of the self loops. SHARPE will internally calculate the probability associated with the self loop.

4.2 Semi-markov chains

SHARPE allows two kinds of semi-Markov chains: acyclic and irreducible.

An acyclic semi-Markov chain is specified as follows:

> **semimark** *name* {(*param_list*)} { **cond** | **uncond** }
> ∗ section 1: transitions and transition distributions
> <*name name dist*>
> ∗ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**
> ∗ section 3: initial state probabilities
> <*name expression*>
> **end**

The specification is the same as for Markov chains with absorbing states, except that instead of instantaneous transition rates we have distribution specifications.

By default, the distribution associated with a transition is conditional. That is, if F(t) is attached to the transition from state A to state B, then F(t) is the probability that the time from entering state A to entering state B is less than or equal to t, given that a state transition from A to B takes place.

The default can be overridden on the command line; the flag **-su** (for semi-Markov unconditional) causes the default to be to interpret the distributions to be unconditional. That is, F(t) is the unconditional probability that the time from entering A to entering B is less than or equal to t. The function F(t) is thus defective (less than one as t approaches ∞) unless B is the only possible successor of A.

For the sake of completeness, SHARPE also recognizes the flag **-sc** (for semi-Markov conditional), even though it is the default.

The first line of the specification may optionally end in the keyword **cond** or **uncond**. If the line ends in **cond**, all distributions in the specification are interpreted to be conditional, regardless of the default. If the line ends in **uncond**, all distributions are interpreted as conditional.

An irreducible semi-Markov chain is specified as follows:

> **semimark** *name* {(*param_list*)} { **cond** | **uncond** }
> ∗ section 1: transitions and transition distributions
> <*name name dist*>
> ∗ section 2: rewards (optional)
> { **reward** {**default** *expression*}
> <*name expression*> }
> **end**

4.3 Reliability block diagrams

A reliability block diagram is specified by

> **block** *name* {(*param_list*)}
> <*blockline*>
> **end**

A *blockline* has one of the following forms:

comp *name dist*

> This is a basic component. It is assigned a name and a distribution.

parallel *name name name* {*name* ...}

> This represents components combined in parallel. The parallel system is assigned the first name and is composed of the rest of the names. The system must have at least two components.

series *name name name* {*name* ...}

> This represents components combined in series. The series system is assigned the first name and is composed of the rest of the names. The system must have at least two components.

kofn *name1 expression, expression, name2*

> This represents k-out-of-n system having identical components. The system is assigned the name *name1*. The first expression gives k and the second expression gives n. *name2* gives a component or sub-block; the block *name1* is assumed to consist of n identically distributed (independent) copies of *name2*. In order for the system to be operating, k of the components must be operating.

kofn *name1, expression, expression, name name* {*name* ...}

> This represents a k-out-of-n system having possibly different components. The system is assigned the name *name1*. The first expression gives k, and the second expression gives n. The second expression is followed by n names, which give the components comprising the system *name1*. The system is assumed to be configured so that in order for the system to be operating, k of the components must be operating. In general, the components and sub-blocks will not have identical failure time distributions. It is important to note where there are commas on this line and where there are not.

In forms 2 through 5, the names making up the block must already be defined.

4.4 Fault trees

A fault tree is specified by the following:

> **ftree** *name* {(*param_list*)}
> <*ftreeline*>
> **end**

A *ftreeline* has one of the following forms:

basic *name dist*

> This is a basic component. It is assigned a name and a distribution. Whenever this name appears later in the fault tree specification, it is interpreted as being a physically distinct copy of a component having the assigned distribution.

repeat *name dist*

> This is also a basic component assigned a name and a distribution. In this case, whenever this name appears later in the fault tree specification, it is interpreted as being the same physical component.

transfer *name name*

> The first name must have been previously defined using either **basic** or **repeat**. The second name is associated with the first, and whenever the second name appears later in the fault tree specification, it is interpreted as being the same physical component as the first name.

and *name name name* {*name* ...}

> This represents and "and" gate. The gate is assigned the first name, and the rest of the names form the inputs to the gate. There must be n inputs.

or *name name name* {*name* ...}

> This represents and "or" gate. The gate is assigned the first name, and the rest of the names form the inputs to the gate. There must be two inputs.

kofn *name expression, expression, name*

> This represents a k-out-of-n gate having identical inputs. The gate is assigned the first name. The first expression gives k and the second expression gives n. The inputs to the gate are assumed to be n identically distributed, independent copies of the second name.

kofn *name expression, expression, name name* {*name* ...}

This represents a k-out-of-n gate whose inputs are not necessarily identical. The gate is assigned the first name. The first expression gives k and the second expression gives n. The names following the second expression are the inputs to the gate; there must be at least two. The inputs are assumed to be configured so that in order for the system to fail, k of the inputs must fail.

In forms 2 through 5, the names making up the block must already be defined.

4.5 Reliability graphs

A reliability graph is specified by the following:

relgraph *name* {(*param_list*)}
∗ section 1: unidirectional edges
<*edge_name edge_name distribution*>
∗ section 2: bidirectional edges (optional)
{**bidirect**
<*edge_name edge_name distribution*> }
end

In the first section, unidirectional edges are specified. A path exists from the first *edge_name* to the second *edge_name*. The *distribution* is the CDF for the time-to-failure of the path.

In the second section (which is optional), bidirectional edges are specified. Two paths exist, one from the first *edge_name* to the second and one from the second to the first, each having the *distribution* as the time-to-failure CDF.

4.6 Single-chain product-form queueing networks

A single-chain product-form queueing network is specified as follows:

pfqn *name* {(*param_list*)}
∗ section 1: station-to-station probabilities
<*station_name station_name expression*>
end
∗ section 2: station types and parameters
<*station_name station_name expression*, ... >
end
∗ section 3: number of customers per chain
<*chain_name expression*>
end

In the first section, the two names represent station names in the queueing network, and the expressions is the probability that a job goes to the second station after it has been served at the first.

The second section defines the service type and parameters of each station. *station_type* is chosen from a predefined set of types. The number of expressions depends on the server type. The possibilities for the lines in this section are as follows:

1. *station_name* **is** *rate*

   The station is an infinite server; each job at the server has exponential service-time CDF with the specified rate.

2. *station_name* **fcfs** *rate*

   The station is a first-come-first-server server. Jobs in the queue are served one at a time; the job being served (if any) has exponential service-time CDF with the specified rate.

3. *station_name* **ps** *rate*

   Jobs at the station share the server. When n jobs are at the station, each has exponential service-time CDF with rate $(rate/n)$.

4. *station_name* **lcfspr** *rate*

   The serving algorithm is "last come first served, preemptive resume".

5. *station_name* **ms** *number_of_servers*, *rate*

   The station contains multiple servers; the number of servers is given by the *expression number_of_servers.* Each server had the same rate.

6. *station_name* **lds** *rate*, *rate*, ...

   There is one server, whose service rate depends on the number of jobs at the station. The first rate applies when there is one job, the second rate when there are two jobs, and so on. If there are fewer rates given than the maximum number of jobs, the last rate on the line is assigned to all the numbers of jobs for which no rate was explicitly given.

The third section gives the number of customers in the network. Although the network has only a single chain, it is expected that the chain be given a name. The name is never used.

4.7 Multiple-chain product-form queueing networks

A multiple-chain product-form queueing network is specified as follows:

> **mpfqn** *name* {(*param_list*)}
> * section 1: station-to-station probabilities for each chain
> <**chain** *chain_name*
> <*station_name station_name expression*>
> **end**>
> **end**
> * section 2: station types and parameters
> <<*station_name station_type* {*expression*, ...}>

> <*chain_name expression*, ... >
> **end**>
> **end**
> ∗ section 3: number of customers per chain
> <*chain_name expression*>
> **end**

In the first section, the two names represent station names in the queueing network, and the expression is the probability that a job goes to the second station after it has been served at the first.

The second section defines the service type and parameters of each station in each chain. *station_type* is chosen from the same pre-defined set of types as for single-chain product-form queueing networks.

A particular station is assigned one station type (it cannot have different station types per chain). For stations of the multiple server type, the number of servers is the same for all chains.

Except for FCFS stations, stations are allowed to have different rates for each chain. An FCFS station must have its station type and rate specified as follows:

> *station_name* **fcfs** *expression*
> **end**

For other stations, there are two ways to specify the server type and rates. The first way is to specify the rates for each chain, like this:

> *station_name station_type* {*number_of_servers*}
> <*chain_name expression*, ... >
> **end**>

number_of_servers is present if and only if *station_type* is **ms** (multiple server). Multiple rates are expected if and only of *station_type* is **lds** (load-dependent server). A chain-specific line must be present for every chain in the network, even for chains which do not contain the station.

The second way to specify server rates is to specify a default on the line that defines the server type. The default rate (or list of rates) is assigned to the server for all chains. For each chain-specific line following the default, the rates given there override the default for that particular chain. The number of chain-specific lines can be zero. Here is the form using the default:

> *station_name station_type*{*number_of_servers*} *expression*, ...
> <*chain_name expression*, ... >
> **end**>

The third section gives the number of customers in each chain.

4.8 Generalized stochastic petri nets

A generalized stochastic petri net is specified as follows:

> **gspn** *name* (*param_list*)
> ∗ section 1: places and initial numbers of tokens
> <*place_name expression*>
> **end**
> ∗ section 2: timed transition names, types and rates
> <*transition_name* **ind** *expression*>
> <*transition_name* **dep** *place_name expression*>
> **end**
> ∗ section 3: immediate transition names and weights
> <*transition_name* **ind** *expression*>
> <*transition_name* **dep** *place_name expression*>
> **end**
> ∗ section 4: place-to-transition arcs and multiplicity
> <*place_name transition_name expression*>
> **end**
> ∗ section 5: transition-to-place arcs and multiplicity
> <*transition_name place_name expression*>
> **end**
> ∗ section 6: inhibitor arcs and multiplicity
> <*place_name transition_name expression*>
> **end**

Each line in the first section specifies a place name and the initial number of tokens in the place.

Each line in the second section specifies a name for a timed transition, a transition type (**ind** if the transition rate is marking-independent and **dep** if it is marking-dependent), a place name if and only if the rate is dependent, and a rate. If the transition is marking-dependent, the effective rate of the transition depends on (is multiplied by) the number of tokens present in the place.

Each line in the third section specifies a name for an immediate transition, a transition type (**ind** if the transition weight is marking-independent and **dep** if it is marking-dependent), a place name if and only if the weight is dependent, and a weight. If the transition is dependent, the effective weight of the transition depends on (is multiplied by) the number of tokens present in the place. The transition weight determines the probability that the transition is chosen if it is one of multiple immediate transitions leaving a place.

The lines in the fourth section specify the arcs from places to transitions. The multiplicity indicates the number of tokens that must be present in the place for the transition to fire.

The lines in the section five specify the arcs from transitions to places. The multiplicity indicates the number of tokens that are deposited in the place when the transition is fired.

The lines in section six specify inhibitor arcs from places to transitions. The multiplicity indicates how may tokens must be in the place to inhibit the transition from firing.

SHARPE allows gspns to have the same three types it allows for Markov chains: acyclic, irreducible, and PH-type.

4.9 Series_parallel graphs

A series-parallel graph is specified as follows:

> **graph** *name* {(*param_list*)}
> <*name* {*name*}>
> **end**
> <*graphline*>
> **end**

The first group of lines specifies the edges in the graph. The edges do not have to be sorted. There may be more than one start and/or terminating edge. It is possible for a name to appear alone on a line. This represents a node having no predecessors and no successors.

A *graphline* has one of the following forms:

dist *name dist*

> This assigns the given distribution to the given graph node. A distribution must be specified for each graph node.

exit *name exit_type*

> This assigns the given exit type to the given node. For every node that has more than one exiting edge, an exit type must be satisfied. If a graph called g has more than one entrance node (node with no predecessors), then SHARPE supplies a dummy entrance node called $E.g$ with zero distribution and edges leading from $E.g$ to each user-specified entrance node. When this is the case, the user must supply an exit type for the node $E.g$.

prob *name name expression*

> The expression gives a probability value to be assigned to the edge going from the first name to the second name. For each node x that has n successors and whose exit type is **prob**, probability values must be assigned to at least n-1 of the edges leading out of x. If values are given for all of the edges, the sum of the values must be one. If one value is missing, the sum of the values must be less than one and SHARPE will compute the missing value.

multpath

> This line requests multiple-path information for the system. Whenever there are probabilistic subgraphs that are not inside maximum, minimum or k-out-of-n subgraphs, SHARPE considers the graph to contain more than one path.

If multiple-path information is requested, SHARPE will compute for each path the probability of taking the path and the conditional distribution for the time-to-finish, given that the path is taken.

The exit types (*exit_type*) are

prob - The parallel subgraphs are probabilistic.

max - All of the subgraphs must complete before going on.

min - One of the parallel subgraphs must complete before going on.

kofn - The first expression gives k and the second expression gives n. $k$out of the n parallel subgraphs must complete before going on. If this exit type is specified for a graph with exactly one successor node, that node is assumed to be duplicated n times, with each copy being identically distributed. Except for this case, it is required that the node with **kofn** exit type have exactly n following parallel subgraphs.

5 Built in functions

SHARPE provides built-in functions (*built_in_function*) that return information about model specification and values resulting from models analysis. Tables 1-2 show all of the built-in functions currently available. Where applicable, it shows the systems to which each built-in function applies and the value returned.

The functions **value** and **tvalue** provide the same information for Markov chains; the value of each is the value at a specified time t of the transient probability function for being in a specific state (or, if no state is specified, of having reached absorption). However, they arrive at the value in different ways. When **value** is used, SHARPE first computes the transient probability function symbolically in t, then evaluates the function at t. When **tvalue** is used, SHARPE uses a numerical algorithm to compute the value directly. The first method is faster, especially when t is large relative to the rates in the chain. The second method is more stable. For more information about ways in which the user can control what algorithms are used for analysis, see section 7, "Controlling the Analysis Process".

The function resulting from analysis of a system generally has the same interpretation as the functions assigned to components. In Tables 1-2 , it has been assumed that the functions assigned to the system components are CDFs; the result function is again a CDF. If instead, for example, the components of the fault tree F had been assigned availability functions, then **value** (5;F) would be the transient availability of the system at time t=5. It should be noted that **mean** and **variance** may not have a meaningful interpretation for some kinds of component (and result) functions.

State names (*state_eword*) are allowed only for Markov or semi-Markov chains. If a function has an argument list, the argument list must match up with the parameters (possibly none) for the system.

6 Asking for results

6.1 Number of digits printed

The statement
format *constant_expression*
specifies the number of digits after the decimal point to be printed in results. This is no way changes the way calculations are carried out internally. It simply specifies the reported precision on output.

6.2 Format for complex numbers

The keyword **imag** is used to control whether a CDF containing complex numbers is printed in complex-number form or sine-cosine form. The statement **imag on** causes results to be printed in complex-number form. The statement **imag off** causes results to be printed in sine-cosine form.

6.3 Printing a system type

To have SHARPE print the type of a system, use the command

type *system_name*

This can be used, for example, to verify that the type of a chain (acyclic, irreducible or PH-type) is what was intended.

6.4 Verbose Output

SHARPE provides two ways of asking for verbose output. If the flag **-v** is used on the command line, SHARPE turns verbose output on for the entire input file unless it is turned off within the file. To turn verbose output on and off within the file, the following commands are used:

> * the following turns verbose output on
> **verbose on**
> the following turns verbose output off
> **verbose off**

When verbose output is turned on, SHARPE prints the following information:

- for Markov or semi-Markov chains, a list of the absorbing states (if any)
- whenever results for the system are requested, the type of the system and whether or not a new analysis is being done

prob (*system_name, state_eword* {;*arg_list*})	
chain with absorbing states	the probability of visiting the state
irreducible Markov chain	the steady state probability of being in the state
mean (*system_name* {,*state_eword*} {;*arg_list*})	
graph	mean of the traversal-time
block diagram and fault tree	mean of the failure time
reliability graph	mean of the failure time
chain with absorbing states, no state given	mean time to absorption
chain with absorbing states, absorbing state given	mean time to reach the state, given that the state is reached
chain with absorbing states, transient state given	mean of unconditional time from start to leaving the state
non-irreducible gspn	mean time to absorption
variance (*system_name* {,*state_eword*} {;*arg_list*})	
same as for **mean**	variance of the CDF
pzero (*system_name* {,*state_eword*} {;*arg_list*})	
same as for **mean** but not gspn	mass at zero of the CDF
pinf (*system_name* {,*state_eword*} {;*arg_list*})	
same as for **mean** but not gspn	mass at infinity of the CDF
pcont (*system_name* {,*state_eword*} {;*arg_list*})	
same as for **mean** but not gspn	"continuous" part of the CDF
value (*t*; *system_name* {,*state_eword*} {;*arg_list*})	
graph	value of t of the traversal-time CDF
block diagrams and fault trees	value of t of the failure time CDF
reliability graph	value of t of the failure time CDF
chain with absorbing states, no state given	value of t of CDF for time to absorption
chain with absorbing states, absorbing state given	value of t of CDF for time to reach the state, given that the state is reached
chain with absorbing states, transient state given	value of t of transient probability function for being in the state
non-irreducible gspn	value of t of CDF for time to absorption
tvalue (*t*; *system_name* {,*state_eword*} {;*arg_list*})	
phase-type Markov chain, no state given	transient probability of having reached absorption at time t
Markov chain (all types), state given	transient probability of being in the state at time t
sreward (*system_name* {,*state_eword*} {;*arg_list*})	
chain with rewards	reward rate assigned to the state
exrss (*system_name* {;*arg_list*})	
irreducible chain with rewards	expected steady state reward rate
exrt (*t*; *system_name* {;*arg_list*})	
non-irreducible chain with rewards	expected reward rate at time t
cexrt (*t*; *system_name* {;*arg_list*})	
non-irreducible chain with rewards	cumulative expected reward over $(0,t)$
rvalue (*r*; *system_name* {;*arg_list*})	
non-irreducible chain with rewards	probability that cumulative reward is less than r when an absorbing state is reached
sum (*index, low, high, expression*)	
not applicable	sum for *index* from low to high by 1 of the expression

Table 1: Built-in Functions

tput (*system_name, eword* {;*arg_list*}) pfqn or gspn	through put for a single-chain pfqn station or gspn transition
rtime (*system_name, station_eword* {;*arg_list*}) pfqn	average response time of a single-chain pfqn station
qlength (*system_name, station_eword* {;*arg_list*}) pfqn	average queue length of a single-chain pfqn station
util (*system_name, eword* {;*arg_list*}) pfqn or gspn	utilization for a single-chain pfqn station or gspn transition
mtput (*system_name, station_eword*{,*chain_eword*} {;*arg_list*}) multiple-chain pfqn, chain given multiple-chain pfqn, no chain given	throughput for a multi-chain pfqn station within a chain throughput for a multi-chain pfqn station, sum over all chains
mrtime (*system_name, station_eword*{,*chain_eword*} {;*arg_list*}) multiple-chain pfqn, chain given multiple-chain pfqn, no chain given	average response time for a multi-chain pfqn station within a chain average response time for a multi-chain pfqn station, sum over all chains
mqlength (*system_name, station_eword*{,*chain_eword*} {;*arg_list*}) multiple-chain pfqn, chain given multiple-chain pfqn, no chain given	average queue length for a multi-chain pfqn station, within a chain average queue length for a multi-chain pfqn station, sum over all chains
mutil (*system_name, station_eword*{,*chain_eword*} {;*arg_list*}) multiple-chain pfqn, chain given multiple-chain pfqn, no chain given	utilization for a multi-chain pfqn station within a chain utilization for a multi-chain pfqn station, sum over all chains
etok (*system_name, place_eword* {;*arg_list*}) gspn	average number of tokens in the place
prempty (*system_name, place_eword* {;*arg_list*}) gspn	probability that the place is empty
etokt (*t*; *system_name, place_eword* {;*arg_list*}) non-irreducible gspn	expected number of tokens in the place at time *t*
premptyt (*t*; *system_name, place_eword* {;*arg_list*}) non-irreducible gspn	probability that the place is empty at time *t*
tputt (*t*; *system_name, transition_eword* {;*arg_list*}) non-irreducible gspn	throughput of the transition at time *t*
utilt (*t*; *system_name, transition_eword* {;*arg_list*}) non-irreducible gspn	utilization of the transition at time *t*
tavetokt (*t*; *system_name, place_eword* {;*arg_list*}) non-irreducible gspn	time-averaged number of tokens in the place during (0,*t*)
tavtputt (*t*; *system_name, transition_eword* {;*arg_list*}) non-irreducible gspn	time-averaged throughput of a transition during (0, *t*)

Table 2: More Built-in Functions

system type	component function meaning	result function meaning
acyclic Markov chain	transition rate	CDF of time to absorption
PH-type Markov chain	transition rate	CDF of time to absorption
acyclic semi-Markov chain	transition distribution	CDF of time to absorption
reliability block diagram	component time-to-failure CDF	system time-to-failure
reliability block diagram	component availability	system availability
reliability graph	time-to-failure CDF	system time-to-failure
fault tree	component time-to-failure CDF	system time-to-failure
fault tree	component failure probability	system failure probability
series-parallel graph	component time-to-completion CDF	system time-to-completion CDF
non-irreducible gspn	transition rate	CDF of time to absorption

Table 3: Definition of **cdf**

- for gspns the type (acyclic, irreducible or PH-type) of gspn just after the gspn specification is read
- for reliability graphs, a list of paths from source to sink
- when a PH-type chain or gspn is analyzed, the algorithm and methods therein are used (see section 7.1)
- when the "new" algorithm is being used for a PH-type chain or gspn, the condition number of the underlying matrix
- for gspns, assorted cryptic information useful for debugging purposes
- when the uniformization algorithm is used, the values of **l** (left truncation point) and **k** (right truncation point)
- when the uniformization algorithm is used with steady-state checking, the value of **l** or **k** when steady state is reached (if at all)
- warnings whenever adjustments are made because of numerical considerations

6.5 Printing results of model analysis

SHARPE provides the following statements for printing the results of analyzing models.

cdf (*system_name* {;*arg_list*})

This statement asks for the function computed when a system is analyzed. Usually this is a cumulative distribution function (CDF), but it may have another interpretation, depending on the system and the meaning of the functions assigned to the system components. The mean and variance of the function are also printed. Tables 3 shows the legal system types and the meaning of the result function for common meanings of the component functions.

cdf (*chain_name, state_eword* {;*arg_list*})

If the specified system was a graph and multiple-path information was requested, the CDF is given for each path.

This statement asks for information pertaining to a particular state in Markov or semi-Markov chain. The only types of chains allowed are acyclic Markov, PH-type Markov and acyclic semi-Markov. If the state is absorbing, the function printed is the CDF (and its mean and variance) of the time to reach that state, conditional on reaching the state. If the state is not absorbing, SHARPE prints the transient probability function for the state. In either case, SHARPE also prints the probability of visiting the state.

lcdf (*chain_name, state_eword* {;*arg_list*})

This statement asks for information pertaining to a particular state in an acyclic Markov or semi-Markov chain, conditional on reaching the state. SHARPE prints the CDF of the time elapsed from the initial time until leaving the state, conditional on entering the state. SHARPE also prints the mean and variance of the CDF and the probability of visiting the state.

reward (*chain_name* {;*arg_list*})

This statement is legal only for Markov and semi-Markov chains with absorbing states and rewards. The function printed is R(r), the probability that the accumulated reward at time of absorption is less than or equal to r.

pqcdf (*system_name* {;*arg_list*})

This statement is legal for fault trees and reliability graphs. SHARPE prints the time-to-failure CDF for the fault tree symbolically in terms of the time-to-failure functions of the individual components or edges. The result consists of a sum of products where the multiplicands are functions Q_i and P_i, where $Q_i(t)$ is the time-to-failure CDF of component or edge i and P_i is $1 - Q_i$.

eval (*system_name* {;*state_eword*} {;*arg_list*}) *low high increment* {*function*}

The arguments are the same as for **cdf**. *low*, *high* and *increment* are all *expressions*. *function* is **cdf**, **reward**, or any of the built-in-function names that take a time parameter. If no *function* appears, the default is **cdf**.

SHARPE evaluates *function* over the interval (*low*, *high*) at increments of *increment*. If the specified system was a graph and multiple-path information was requested, the evaluation is given for each path.

expr *expression* {,*expression* ... }

SHARPE prints the value of the expression(s).

6.6 Using a loop to print results

A *loop* may be used for printing results; the only legal statements within a loop are **expr**, **loop**, **bind** and **epsilon** statements. See section 7.3 for information about the **epsilon** statement. The syntax is

loop *simple_var,low, high*{,*increment*}
< *loop* >
<**bind** *simple_var expression*>
<**expr** *expression* {,*expression* ... }>

<epsilon *e_type* **expression>**
end

low, *high* and *increment* may be any *expressions*. If no increment is present, it is assumed to be one. The statement types within the loop may be intermixed.

7 Controlling the analysis process

This section describes the statements SHARPE provides to allow the user to exercise some options for how model analysis is done.

7.1 Phase-type markov chain analysis

SHARPE provides three algorithms for analyzing phase-type Markov chains: *old*, *new* and *uniformization*. The *old* and *new* algorithms are "symbolic algorithms", producing functions symbolic in the time variable t. Their names are derived from the fact that one algorithm (*old*) was implemented before the other (*new*). The *old* algorithm is $O(n_5)$ and the *new* is $O(n_3)$. However, the *old* algorithm appears to be more accurate and the difference in efficiency is small for small chains. The *uniformization* algorithm requires a particular time t and computes transient probabilities for that value of t.

When results are requested using **cdf** or the built-in-function **value**, one of the two symbolic algorithms is used. (In the case of **value**, the CDF is computed and then evaluated at **t**) By default, the *old* symbolic algorithm is used. The statements **phnew** and **phold** control explicitly which symbolic algorithm is used. When **phnew** appears, SHARPE switches to using the *new* algorithm for all **cdf** and **value** requests it sees. The statement **phold** causes SHARPE to switch back to using the *slow* algorithm.

It is possible to change the default algorithm on the command line. If the flag **-pn** is used, the default algorithm is *new*. the algorithm can still be changed within the file by using **phold** and **phnew**. For the sake is completeness, the flag **-po** to make *old* the default, is recognized even though *old* is the default anyway.

Both of the symbolic algorithms require as one of their steps finding the eigenvalues of a matrix. SHARPE contains two routines for finding eigenvalues. The first was included when the *old* symbolic algorithm was implemented; the second came with the *new* symbolic algorithm. The main difference between the two is that the second eigenvalue finder leaves the input matrix in upper Hessenberg form.

The *old* algorithm can use either eigenvalue finder (it uses the first algorithm by default), but the *new* algorithm requires the second eigenvalue finder. When SHARPE sees **phnew** or the flag **-pn**, it automatically uses the second eigenvalue finder. The SHARPE statements **eigen1** and **eigen2** control which eigenvalue finder is used. If **eigen1** is used after the *new* algorithm has been specified, **eigen2** will be ignored.

The eigenvalue finder can be specified on the command line. The flag **-e1** chooses the first eigenvalue finder; **-e2** chooses the second. **eigen1** and **eigen2** statements in an input file override these command line options.

Both of the symbolic algorithms require as one of their steps choosing a set of complex values for a variable. There are two methods available, *simple* and *voronoi*. The *voronoi* method attempts to choose the values in such a way as to improve the numerical stability of the overall algorithm for analyzing phase-type chains. Either value-choosing method can be used with the *new* symbolic algorithm; the *old* algorithm can only use the *simple* method. By default, both algorithms use the *simple* method.

The statements **voronoi on** and **voronoi off** control which value-choosing method is used. When SHARPE sees **phold** or the flag **-po**, it automatically uses the *simple* method. An attempt to choose the voronoi method after the slow algorithm has been chosen will be ignored. The flag **-vo** on the SHARPE command line causes the *voronoi* method to be used. The statements **voronoi on** and **voronoi off** override the command line flag.

If SHARPE is run with verbose output turned on, it prints which symbolic algorithm, which eigenvalue finding method, and which whenever it analyzes a phase-type Markov chain for **cdf** or **value**.

The *uniformization* algorithm is chosen using the built-in function **tvalue**. This algorithm's efficiency is linear in time parameter t, but the constant is such that it may take quite a long time for large t.

The *uniformization* algorithm can optionally check, during its iterations, whether t is large enough that the chain has reached steady state. The check itself is time-consuming, so SHARPE allows the check to be turned on and off. The statement **unif_ss on** turns the check on; **unif_ss off** turns it off. The default is for steady-state checking to be off. Steady-state checking can also be turned on on the command line by using the flag -ss. The flag is overridden by **unif_ss** statements inside input files.

Table 4 summarizes the possible combinations of algorithms and method within algorithms that are available and indicates the statements that would be used to have SHARPE use each combination. When a statement appears in parentheses in the tabel, it means that the statement can be omitted because it is the default.

7.2 Irreducible markov and semi-markov chain analysis

SHARPE contains two algorithms for obtaining steady-state probabilities for irreducible Markov and semi-Markov chains: SOR and Gauss-Seidel. SHARPE always begins its analysis by using the SOR algorithm. If SOR does not converge to a solution after a certain number of iterations, SHARPE prints the number of iterations and the "tolerance" and asks whether to continue with SOR, switch to Gauss-Seidel, or terminate the algorithm. The "tolerance" is the ratio between the largest (in absolute value) value in the most recent probability vector.

If the choice is to continue, SHARPE will do so and will continue to prompt for a decision every time a certain number of iterations has gone by without convergence. As long as SOR is still being used, the options are to continue SOR, switch to Gauss-

Combination of Methods			
algorithm	eigenvalue finder	value chooser	statements
slow	first	simple	(**phold**) (**eigen1**) (**voronoi off**) **cdf** or **value**
slow	second	simple	(**phold**) **eigen2** (**voronoi off**) **cdf** or **value**
fast	second	simple	**phnew** (**eigen2**) (**voronoi off**) **cdf** or **value**
fast	second	voronoi	**phnew** (**eigen2**) **voronoi on** **cdf** or **value**
uniformization without steady state	NA	NA	(**unif_ss off**) **tvalue**
uniformization with steady state	NA	NA	**unif_ss on** **tvalue**

Table 4: Available Phase-type Analysis Choices.

Seidel or stop. Once Gauss-Seidel is being used, the only options are to continue Gauss-Seidel or to stop.

If the flag **-f** appears on the command line, SHARPE will start with the SOR algorithm and automatically switch to Gauss-Seidel if SOR has not converged in a certain number of iterations. If **-f** is used, there will be no user interaction required while SHARPE is running. However, it is possible that the algorithm might not terminate on certain models.

The "certain number of iterations" depends on the number of states in the chain and on whether the **-f** flag is used. If **-f** is used, SHARPE will run the SOR algorithm for more iterations than the number that would cause the first user prompt without the **-f** flag.

SHARPE allows the **tvalue** built-in function to be applied to irreducible Markov chains to obtain transient state probabilities using the uniformization algorithm. If the time **t** is large enough, **tvalue** will find steady-state probabilities, but the algorithm can be very time-consuming in this case.

7.3 Values of epsilon

SHARPE contains five user-controlled "epsilons", the small values that determine when algorithms have converged or when two floating point numbers are equal. To set the value for one of these epsilons, the following statement is used:

epsilon *epsilon_id expression*

where *epsilon_id* is one of the following:

basic - this sets the value that determines when two floating point numbers are equal or when a floating point number is zero.

uniform - this epsilon determines when the uniformization algorithm has converged.

findeigen - This determines when either of the two eigenvalue-finding algorithms has converged.

sorteigen - Both the old and new symbolic algorithms for phase-type chains must sort eigenvalues after they are found. This epsilon determines when two eigenvalues are considered to be equal. This is very important to the slow symbolic algorithm, since it handles equal eigenvalues as a special case in the remainder of the algorithm.

results - This determines when a printed result is considered to be zero

Th statement **info epsilons** can be used to print the current value for all of the "epsilons".

8 Input size limitations

SHARPE contains some static arrays that limit the input size. The following statements ask SHARPE to print its:

info constants

If any of these limits is exceeded, SHARPE prints an error message and stops. Here is a list of the quantities that are limited:

- length of an input line
- number of models
- number of component names (total, overall models)
- number of nodes in a series-parallel, acyclic graph
- number of places in a Petri net
- number of transitions in a Petri net
- number of arcs in a Petri net
- number of symbols (variable, function and distribution names)
- number of states in a Markov chain
- number of edges in a Markov chain
- number of chains in a queueing network

- number of intervals in an **eval** statement
- stack size for internal expression evaluation algorithm
- stack size for internal algorithm for fault trees with repeated components and reliability graphs

9 Summary of top level input statements

A SHARPE input file has the form:

> <*statement*>
> **end**

where *statement* is one of the following (for multiple-line statements, only the first line is shown):

> **var** *name expression*
> **func** *func_name* (*param_list*) *expression*
> **poly** *name* (*param_list*) *dist*
> **graph** *name* {(*param_list*)}
> **markov** *name* {(*param_list*)} {**readprobs**}
> **semimark** *name* {(*param_list*)} {**cond** | **uncond**}
> **block** *name* {(*param_list*)}
> **relgraph** *name* {(*param_list*)}
> **ftree** *name* {(*param_list*)}
> **pfqn** *name* {(*param_list*)}
> **gspn** *name* (*param_list*)
> **bind**
> **bind** *name expression*
> **format expression**
> **cdf** (*system_name*, {*state_eword*} {;*arg_list*})
> **lcdf** (*chain_name*, *state_eword* {;*arg_list*})
> **reward** (*chain_name* {;*arg_list*})
> **pqcdf** (*system_name* {;*arg_list*})
> **eval** (*system_name*, {,*state_eword*} {;*arg_list*}) *low high increment* {*function*}
> **expr** *expression* {,*expression* ... }
> **epsilon** *epsilon_id expression*
> **imag** [**on** | **off**]
> **verbose** [**on** | **off**]
> **unif_ss** [**on** | **off**]
> **voronoi** [**on** | **off**]
> **loop** *low,high*{,*increment*}
> **eigen1**
> **eigen2**
> **phold**
> **phnew**

type *system_name*
info [**epsilons** | **constants**]

10 Examples

In this section, we give a series of examples showing the features of the SHARPE language. It is assumed that this package is used in concert with the book

> Kishor S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications." Prentice Hall, 1982.

which is henceforth denoted as KST. Many of the examples provided in this section are taken directly from this book so the descriptions are very brief but the page number in KST is provided.

The results are presented in tabular form with the input to SHARPE in the left-hand column and the output from SHARPE in the right-hand column of the table. Refer to Appendix A for detailed instructions on using SHARPE.

10.1 Discrete-Time Markov Chain

This is a memory interference model with state diagram on page 328, book by KST. In this example there are two processors and two memory modules for which the processors contend. It is finite state, discrete-time markov chain (DTMC). States (1,1), (0,2) & (2,0) are denoted as 11, 02 and 20 respectively. The transition probabilities q1 and q2 have been chosen randomly. This example uses the built-in function **prob** to find the steady state probabilities of visiting the states of the system. We emphasize that SHARPE can perform a steady-state analysis of DTMCs but it cannot carry out a transient analysis of DTMCs. While specifying a DTMC, make sure that you do not specify the self loop on a state; SHARPE will automatically compute the self loop probability based on the sum of the probabilities to other states.

The following input file to SHARPE for the Discrete-Time Markov Chain Memory Interference model can be run:

```
markov mem_interfere
11 02 q2*q2
11 20 q1*q1
02 11 q1
20 11 q2
end

bind
q1 .6
q2 .4
end
```

```
expr prob(mem_interfere,11)
expr prob(mem_interfere,02)
expr prob(mem_interfere,20)

end
```

Produces the following output:

```
 prob(mem_interfere,11):   4.6154e-01

------------------------------------------

 prob(mem_interfere,02):   1.2308e-01

------------------------------------------

 prob(mem_interfere,20):   4.1538e-01
```

10.2 Markov chain with absorbing state

This is the solution of the finite-state, continuous-time markov chain (CTMC) for the program execution model on page 352 of KST. Note that the text example is a DTMC but since we are interested in carrying out a transient analysis, we change the problem and consider the model to be a CTMC. This example has one absorbing state F, which is denoted as s5 in KST.

The following is the input to SHARPE for a CTMC with absorbing state - Program Execution model:

```
markov pgm_exec


s1 s2 0.6
s1 s3 0.4
s3 s2 0.2
s2 s4 0.6
s3 s4 0.4
s4 s3 0.6
s2 F  0.4
s3 F  0.4
s4 F  0.4
end


end


cdf(pgm_exec,F)
```

```
end
```

Which produces the following result:

```
information about system pgm_exec node F

probability of entering node: 1.0000e+00

conditional CDF for time of reaching this absorbing state

   1.0000e+00 t(  0) exp( 0.0000e+00 t)
+ -1.6667e+00 t(  0) exp(-4.0000e-01 t)
+  6.6667e-01 t(  0) exp(-1.0000e+00 t)

mean: 3.5000e+00
variance: 7.2500e+00
```

10.3 Discrete time uniprogrammed computer

This example solves the uniprogrammed computer system model with m I/O devices and a CPU. The system description and the model is given on page 322 of KST. This model is an irreducible finite-state, discrete-time Markov chain. The number of I/O devices has been arbitrarily chosen to be 5, the probabilities of accessing the devices have been arbitrarily chosen, and the program will continue execution with a probability of 0.5. In state 0 the program is executing on the CPU. The request for I/O device i occurs with a probability "qi" (i is the subscript here). If I/O device i is being accessed/used then the system is in state i, $i > 0$.

The following is the input file to SHARPE for a Uniprogrammed Computer System model:

```
markov unipgmmed_cmpter
0 1 q1
0 2 q2
0 3 q3
0 4 q4
0 5 q5
1 0 1
2 0 1
3 0 1
4 0 1
5 0 1
end

bind
q1 .06
q2 .1
q3 .1
q4 .2
q5 .04
end
```

```
expr prob(unipgmmed_cmpter,0)
expr prob(unipgmmed_cmpter,1)
expr prob(unipgmmed_cmpter,2)
expr prob(unipgmmed_cmpter,3)
expr prob(unipgmmed_cmpter,4)
expr prob(unipgmmed_cmpter,5)

end
```

Which produces the following output:

```
 prob(unipgmmed_cmpter,0):   6.6667e-01

--------------------------------------------

 prob(unipgmmed_cmpter,1):   4.0000e-02

--------------------------------------------

 prob(unipgmmed_cmpter,2):   6.6667e-02

--------------------------------------------

 prob(unipgmmed_cmpter,3):   6.6667e-02

--------------------------------------------

 prob(unipgmmed_cmpter,4):   1.3333e-01
```

10.4 Acyclic Markov Chain - Example 1

In this example, the following language features are used: Markov chain specification, user-defined functions, variable bindings, cdf to print results, eval to print results, built-in functions value and sum, evaluated names, and loop to print results. The input to SHARPE is:

```
* function definitions
func c(x,k) 360 / (360 + (k-1) * x)
func l(x,k) k * x * c(x,k)
func fl(x,k) k * x * (1 - c(x,k))

* acyclic Markov chain & cdf(main,F)

markov chain
5-3 4-3 l(lambda, 5)
4-3 3-3 l(lambda, 4)
3-3 2-3 l(lambda, 3)

5-3 F fl(lambda, 5) + fl(mu, 3)
```

```
4-3 F fl(lambda, 4) + fl(mu, 3)
3-3 F fl(lambda, 3) + fl(mu, 3)
2-3 F (2  * lambda) + fl(mu, 3)

5-3 5-2 l(mu, 3)
4-3 4-2 l(mu, 3)
3-3 3-2 l(mu, 3)
2-3 2-2 l(mu, 3)

5-2 4-2 l(lambda, 5)
4-2 3-2 l(lambda, 4)
3-2 2-2 l(lambda, 3)

5-2 F (2  * mu) + fl(lambda, 5)
4-2 F (2  * mu) + fl(lambda, 4)
3-2 F (2  * mu) + fl(lambda, 3)
2-2 F (2  * mu) + (2  * lambda)
end

* No initial state probabilities are
* given. It will be assumed that state
* 5-3, the only state with no
* predecessors, has probability one and
* all other states have probability zero

end

bind lambda .0001 mu .00001 end
* the following function of t gives
* the probability that state F is reached
* in time <= t

cdf(main,F)

* the following statement asks to
* the following statement asks to
* have cdf(main,F) evaluated at values
* of t starting at 5 and going up to
* 15 by increments of 1

eval(main,F) 5 15 1

* value (t;main, state0 gives the
* transient probability of being in the
* state at time t. The function gp adds
* these for all of the states. We expect
* gp(t) to be 1 for all t

* func gp(t) sum(i,2,3,\
sum(j,2,5, value(t; main,$(j)-$(i)))\
+ value(t; main, F)


* the following loop tries sum values
```

```
of t

loop t,0.5,1,0.1
expr gp(t)
end
end
```

with the output:

```
information about system main node  F

probability of entering node: 1.0000e+00

conditional CDF for time of reaching this absorbing state

   1.0000e+00 t( 0) exp( 0.0000e+00 t)
+ -3.0000e+01 t( 0) exp(-2.2000e-04 t)
+  2.0000e+01 t( 0) exp(-2.3000e-04 t)
+  6.0000e+01 t( 0) exp(-3.2000e-04 t)
+ -4.0000e+01 t( 0) exp(-3.3000e-04 t)
+ -4.5000e+01 t( 0) exp(-4.2000e-04 t)
+  3.0000e+01 t( 0) exp(-4.3000e-04 t)
+  1.2000e+01 t( 0) exp(-5.2000e-04 t)
+ -8.0000e+00 t( 0) exp(-5.3000e-04 t)

mean: 1.2512e+04
variance: 4.3615e+07

------------------------------------------

system main
 node F
  t F(t)
5.0000 e+00 1.0284 e-08
6.0000 e+00 1.4141 e-08
7.0000 e+00 1.8597 e-08
8.0000 e+00 2.3654 e-08
9.0000 e+00 2.9310 e-08
1.0000 e+00 3.5567 e-08
1.1000 e+00 4.2423 e-08
1.2000 e+00 4.9880 e-08
1.3000 e+00 5.7938 e-08
1.4000 e+00 6.6596 e-08
1.5000 e+00 7.5854 e-08

t = 0.500000
gp(t): 1.0000e+00

t = 0.600000
gp(t): 1.0000e+00

t = 0.700000
gp(t):1.0000e+00
```

```
t = 0.800000
gp(t):1.0000e+00

t = 0.900000
gp(t):1.0000e+00

t = 0.100000
gp(t):1.0000e+00
```

10.5 Acyclic Markov Chain - Example 2

This example shows two different ways of specifying transition rate functions of a Markov chain. The following summarizes the input to SHARPE and the resulting output. This a markov chain with absorbing states. F is the final state, i.e., the absorbing state.

```
var c8 360 / (360 + 7 * lambda)
var fc8 1 - c8
var l8 8 * lambda * c8
var fl8 8 * lambda * fc8

markov m1
8-2 7-2 l8
8-2 F   (2 * mu) + fl8
end
end

bind
lambda .0001
mu .00001
end

cdf (m1,F)
end
```

with the resulting output of:

```
information about system m1 node F

probability of entering node: 2.4392e-02

conditional CDF for time of reaching this absorbing state

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-8.2000e-04 t)

mean: 1.2195e+03
variance: 1.4872e+06
```

Another way of defining the above chain is:

```
func c(x, k) 360 / (360 + (k-1) * x)
func l(x, k) k * x * c(x, k)
func fl(x,k) k * x * (1 - c(x, k))

markov m2
8-2 7-2 l(lambda, 8)
8-2 F   (2 * mu) + fl(lambda, 8)
end
end

bind
lambda .0001
mu .00001
end

cdf (m2,F)
end
```

which produces the following results

```
information about system m2 node     F

probability of entering node: 2.4392e-02

conditional CDF for time of reaching this absorbing state

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-8.2000e-04 t)

mean: 1.2195e+03
variance: 1.4872e+06
```

10.6 Reward Distribution

The following illustrate two examples that test the reward distributions and rvalue for acyclic and PH-type chains. Input to SHARPE:

```
* test of reward distribution and rvalue for
* acyclic Markov and PH-type chains

bind
r1 3
r2 8
r3 6
r4 4
end

markov m
1 2 1
```

```
1 3 2
2 4 3
3 4 4
4 5 5
reward
1 r1
2 r2
3 r3
4 r4
end
end

markov mr
1 2 1/r1
1 3 2/r1
2 4 3/r2
3 4 4/r3
4 5 5/r4
end
end

cdf(m)
* reward(m) and cdf(mr) should be equal
reward(m)
cdf(mr)
* test that chain is re-analyzed for ordinary cdf
cdf(m)
* the two values should be equal
expr value(.2;mr),rvalue(.2;m)

bind
ra 3
rb 4
end

markov m2
a b 3
b a 4
a c 9
reward
a ra
b rb
end
a 1
end

markov mr2
a b 3/ra
b a 4/rb
a c 9/ra
end
a 1
end
```

```
cdf(m2)
reward(m2)
cdf(mr2)
cdf(m2)
expr value(.2;mr2), rvalue(.2;m2)
eval (m2) 0 .4 .2 reward
end
```

with the output:

```
CDF for system m:

   -2.5000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.2500e+00 t(  0) exp(-3.0000e+00 t)
 +  1.0000e+01 t(  0) exp(-4.0000e+00 t)
 + -4.7500e+00 t(  0) exp(-5.0000e+00 t)

mean: 8.1111e-01
variance: 2.3136e-01

-------------------------------------------


reward cdf for chain m:

    1.0000e+00 r(  0) exp( 0.0000e+00 r)
 + -7.6190e-01 r(  0) exp(-3.7500e-01 r)
 + -4.2857e+00 r(  0) exp(-6.6667e-01 r)
 +  7.6667e+00 r(  0) exp(-1.0000e+00 r)
 + -3.6190e+00 r(  0) exp(-1.2500e+00 r)

mean: 3.6889e+00
variance: 5.8128e+00

-------------------------------------------


CDF for system mr:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -7.6190e-01 t(  0) exp(-3.7500e-01 t)
 + -4.2857e+00 t(  0) exp(-6.6667e-01 t)
 +  7.6667e+00 t(  0) exp(-1.0000e+00 t)
 + -3.6190e+00 t(  0) exp(-1.2500e+00 t)

mean: 3.6889e+00
variance: 5.8128e+00

-------------------------------------------


CDF for system m:
   -2.5000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.2500e+00 t(  0) exp(-3.0000e+00 t)
 +  1.0000e+01 t(  0) exp(-4.0000e+00 t)
 + -4.7500e+00 t(  0) exp(-5.0000e+00 t)

mean: 8.1111e-01
variance: 2.3136e-01

-------------------------------------------

 value(.2;mr):   8.2368e-04
rvalue(.2;m):   8.2368e-04

-------------------------------------------


CDF for system m2:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.0551e-01 t(  0) exp(-2.7085e+00 t)
 + -5.9449e-01 t(  0) exp(-1.3292e+01 t)

mean: 1.9444e-01
variance: 7.9475e-02

-------------------------------------------


reward cdf for chain m2:

    1.0000e+00 r(  0) exp( 0.0000e+00 r)
 + -3.6132e-01 r(  0) exp(-6.9722e-01 r)
 + -6.3868e-01 r(  0) exp(-4.3028e+00 r)

mean: 6.6667e-01
variance: 1.1111e+00

-------------------------------------------


CDF for system mr2:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -3.6132e-01 t(  0) exp(-6.9722e-01 t)
 + -6.3868e-01 t(  0) exp(-4.3028e+00 t)

mean: 6.6667e-01
variance: 1.1111e+00

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

```
CDF for system m2:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.0551e-01 t(  0) exp(-2.7085e+00 t)
 + -5.9449e-01 t(  0) exp(-1.3292e+01 t)

mean: 1.9444e-01
variance: 7.9475e-02

-------------------------------------------

 value(.2;mr2):   4.1559e-01
 rvalue(.2;m2):   4.1559e-01

-------------------------------------------

      system m2
      r         F(r)

 0.0000 e+00  0.0000 e+00
 2.0000 e-01  4.1559 e-01
 4.0000 e-01  6.1237 e-01
```

10.7 Markov Chain with absorbing states

The following example uses the built-in functions cdf and lcdf. It demonstrates the use of value to check that the system is in any one of the states specified. This can be helpful in validating the model.

The following is the input to SHARPE:

```
markov m
1 2 1
1 3 2
2 4 3
3 4 4
4 5 5
end
end

lcdf(m,1)
lcdf(m,2)
lcdf(m,3)
lcdf(m,4)
cdf(m,5)
cdf(m,1)
cdf(m,2)
cdf(m,3)
cdf(m,4)
expr value(.1;m,1),value(.1;m,2),value(.1;m,3),\
  value(.1;m,4),value(.1;m,5)
expr value(.1;m,1)+value(.1;m,2)+value(.1;m,3)+\
  value(.1;m,4)+value(.1;m,5)
expr value(.2;m,1)+value(.2;m,2)+value(.2;m,3)+\
  value(.2;m,4)+value(.2;m,5)
expr value(.5;m,1)+value(.5;m,2)+value(.5;m,3)+\
  value(.5;m,4)+value(.5;m,5)

markov m2
1 2 1
1 3 2
A 2 8
2 4 3
3 4 4
4 5 5
end
1 .4
A .4
2 .2
end

lcdf(m2,1)
lcdf(m2,2)
lcdf(m2,3)
lcdf(m2,4)
lcdf(m2,A)
cdf(m2,5)
cdf(m2,1)
cdf(m2,2)
cdf(m2,3)
cdf(m2,4)
cdf(m2,A)
expr value(.1;m2,1),value(.1;m2,2),value(.1;m2,3)
expr value(.1;m2,4),value(.1;m2,5),value(.1;m2,A)
expr value(.1;m2,1)+value(.1;m2,2)+value(.1;m2,3)+\
  value(.1;m2,4)+value(.1;m2,5)+value(.1;m2,A)
expr value(.2;m2,1)+value(.2;m2,2)+value(.2;m2,3)+\
  value(.2;m2,4)+value(.2;m2,5)+value(.2;m2,A)
expr value(.5;m2,1)+value(.5;m2,2)+value(.5;m2,3)+\
  value(.5;m2,4)+value(.5;m2,5)+value(.5;m2,A)

markov mm
a b 3
a c 9
a d 4
end
end

cdf(mm,a)
cdf(mm,b)
cdf(mm,c)
cdf(mm,d)
expr value(.1;mm,a)+value(.1;mm,b)+\
  value(.1;mm,c)+value(.1;mm,d)
end
```

with the following output:

```
information about system m node     1

probability of entering node: 1.0000e+00

conditional CDF for time of leaving this node

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-3.0000e+00 t)

mean: 3.3333e-01
variance: 1.1111e-01

-------------------------------------------

information about system m node     2

probability of entering node: 3.3333e-01

conditional CDF for time of leaving this node

   -3.0000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-3.0000e+00 t)

mean: 6.6667e-01
variance: 2.2222e-01

-------------------------------------------

information about system m node     3

probability of entering node: 6.6667e-01

conditional CDF for time of leaving this node

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.0000e+00 t(  0) exp(-3.0000e+00 t)
 +  3.0000e+00 t(  0) exp(-4.0000e+00 t)

mean: 5.8333e-01
variance: 1.7361e-01

-------------------------------------------

information about system m node     4

probability of entering node: 1.0000e+00

conditional CDF for time of leaving this node

   -2.5000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.2500e+00 t(  0) exp(-3.0000e+00 t)
 +  1.0000e+01 t(  0) exp(-4.0000e+00 t)
 + -4.7500e+00 t(  0) exp(-5.0000e+00 t)

mean: 8.1111e-01
variance: 2.3136e-01

-------------------------------------------

information about system m node     5

probability of entering node: 1.0000e+00

conditional CDF for time of reaching this absorbing state

   -2.5000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.2500e+00 t(  0) exp(-3.0000e+00 t)
 +  1.0000e+01 t(  0) exp(-4.0000e+00 t)
 + -4.7500e+00 t(  0) exp(-5.0000e+00 t)

mean: 8.1111e-01
variance: 2.3136e-01

-------------------------------------------

information about system m node     1
transient state probability:

    1.0000e+00 t(  0) exp(-3.0000e+00 t)
-------------------------------------------

information about system m node     2
transient state probability:

    1.0000e+00 t(  1) exp(-3.0000e+00 t)
-------------------------------------------

information about system m node     3
transient state probability:

    2.0000e+00 t(  0) exp(-3.0000e+00 t)
 + -2.0000e+00 t(  0) exp(-4.0000e+00 t)
-------------------------------------------

information about system m node     4
transient state probability:

    1.5000e+00 t(  1) exp(-3.0000e+00 t)
 +  3.2500e+00 t(  0) exp(-3.0000e+00 t)
 + -8.0000e+00 t(  0) exp(-4.0000e+00 t)
 +  4.7500e+00 t(  0) exp(-5.0000e+00 t)
-------------------------------------------
```

```
value(.1;m,1):   7.4082e-01
value(.1;m,2):   7.4082e-02
value(.1;m,3):   1.4100e-01
value(.1;m,4):   3.7242e-02
value(.1;m,5):   6.8614e-03

-------------------------------------------


value(.1;m,1)+value(.1;m,2)+value(.1;m,3)+\
  value(.1;m,4)+value(.1;m,5):1.0000e+00

-------------------------------------------


value(.2;m,1)+value(.2;m,2)+value(.2;m,3)+\
  value(.2;m,4)+value(.2;m,5):1.0000e+00

-------------------------------------------


value(.5;m,1)+value(.5;m,2)+value(.5;m,3)+\
  value(.5;m,4)+value(.5;m,5):1.0000e+00

-------------------------------------------

information about system m2 node     1

probability of entering node: 4.0000e-01

conditional CDF for time of leaving this node

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-3.0000e+00 t)

mean: 3.3333e-01
variance: 1.1111e-01

-------------------------------------------

information about system m2 node     2

probability of entering node: 7.3333e-01

conditional CDF for time of leaving this node

   -5.4545e-01 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.3273e+00 t(  0) exp(-3.0000e+00 t)
 +  3.2727e-01 t(  0) exp(-8.0000e+00 t)

mean: 4.6212e-01
variance: 1.5197e-01
```

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

information about system m2 node     3

probability of entering node: 2.6667e-01

conditional CDF for time of leaving this node

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.0000e+00 t(  0) exp(-3.0000e+00 t)
 +  3.0000e+00 t(  0) exp(-4.0000e+00 t)

mean: 5.8333e-01
variance: 1.7361e-01

-------------------------------------------

information about system m2 node     4

probability of entering node: 1.0000e+00

conditional CDF for time of leaving this node

   -1.0000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.6000e+00 t(  0) exp(-3.0000e+00 t)
 +  4.0000e+00 t(  0) exp(-4.0000e+00 t)
 + -4.0000e-01 t(  0) exp(-8.0000e+00 t)

mean: 6.9444e-01
variance: 2.0062e-01

-------------------------------------------

information about system m2 node     A

probability of entering node: 4.0000e-01

conditional CDF for time of leaving this node

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-8.0000e+00 t)

mean: 1.2500e-01
variance: 1.5625e-02

-------------------------------------------

information about system m2 node     5

probability of entering node: 1.0000e+00

conditional CDF for time of reaching this absorbing state
```

```
   -1.0000e+00 t(  1) exp(-3.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -4.6000e+00 t(  0) exp(-3.0000e+00 t)
 +  4.0000e+00 t(  0) exp(-4.0000e+00 t)
 + -4.0000e-01 t(  0) exp(-8.0000e+00 t)

mean: 6.9444e-01
variance: 2.0062e-01

--------------------------------------------

information about system m2 node     1
transient state probability:

    4.0000e-01 t(  0) exp(-3.0000e+00 t)
--------------------------------------------

information about system m2 node     2
transient state probability:

    4.0000e-01 t(  1) exp(-3.0000e+00 t)
 +  8.4000e-01 t(  0) exp(-3.0000e+00 t)
 + -6.4000e-01 t(  0) exp(-8.0000e+00 t)
--------------------------------------------

information about system m2 node     3
transient state probability:

    8.0000e-01 t(  0) exp(-3.0000e+00 t)
 + -8.0000e-01 t(  0) exp(-4.0000e+00 t)
--------------------------------------------

information about system m2 node     4
transient state probability:

    6.0000e-01 t(  1) exp(-3.0000e+00 t)
 +  2.5600e+00 t(  0) exp(-3.0000e+00 t)
 + -3.2000e+00 t(  0) exp(-4.0000e+00 t)
 +  6.4000e-01 t(  0) exp(-8.0000e+00 t)
--------------------------------------------

information about system m2 node     A
transient state probability:

    4.0000e-01 t(  0) exp(-8.0000e+00 t)
--------------------------------------------

 value(.1;m2,1):   2.9633e-01
value(.1;m2,2):   3.6435e-01
value(.1;m2,3):   5.6399e-02

--------------------------------------------

 value(.1;m2,4):   8.3490e-02
value(.1;m2,5):   1.9703e-02
value(.1;m2,A):   1.7973e-01

-------------------------------------------

 value(.1;m2,1)+value(.1;m2,2)+value(.1;m2,3)+\
  value(.1;m2,4)+value(.1;m2,5)+value(.1;m2,A): 1.0000e+00

-------------------------------------------

 value(.2;m2,1)+value(.2;m2,2)+value(.2;m2,3)+\
  value(.2;m2,4)+value(.2;m2,5)+value(.2;m2,A): 1.0000e+00

-------------------------------------------

 value(.5;m2,1)+value(.5;m2,2)+value(.5;m2,3)+\
  value(.5;m2,4)+value(.5;m2,5)+value(.5;m2,A): 1.0000e+00

-------------------------------------------

information about system mm node     a
transient state probability:

    1.0000e+00 t(  0) exp(-1.6000e+01 t)
-------------------------------------------

information about system mm node     b

probability of entering node: 1.8750e-01

conditional CDF for time of reaching this absorbing state

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-1.6000e+01 t)

mean: 6.2500e-02
variance: 3.9062e-03

-------------------------------------------

information about system mm node     c

probability of entering node: 5.6250e-01

conditional CDF for time of reaching this absorbing state

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-1.6000e+01 t)

mean: 6.2500e-02
variance: 3.9062e-03

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

```
information about system mm node     d

probability of entering node: 2.5000e-01

conditional CDF for time of reaching this absorbing state

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.0000e+00 t(  0) exp(-1.6000e+01 t)

mean: 6.2500e-02
variance: 3.9062e-03

--------------------------------------------

value(.1;mm,a)+value(.1;mm,b)+value(.1;mm,c)+\
  value(.1;mm,d):1.0000e+00
```

10.8 Queuing Network

This example is Ex.9.16 of KST. This implements the queueing network ignoring the memory constraint. This corresponds to E[$\hat{\text{R}}$] in table 9.12. The following is the input to SHARPE:

```
bind
p0 0.05
p1 0.5
p2 0.3
p3 0.15
scpu 89.3
sio1 44.6
sio2 26.8
sio3 13.4
sterm 1/15
end
pfqn ex9.16(n)
cpu term p0
cpu io1 p1
cpu io2 p2
cpu io3 p3
io1 cpu 1
io2 cpu 1
io3 cpu 1
term cpu 1
end
cpu fcfs scpu
term is sterm
io1 fcfs sio1
io2 fcfs sio2
io3 fcfs sio3
end
cust n
end
```

```
func ET(N) scpu*util(ex9.16,cpu;N)*p0
func ERhat(M) M/ET(M) - 1/sterm
expr ERhat(10)
expr ERhat(20)
expr ERhat(30)
expr ERhat(40)
expr ERhat(50)
expr ERhat(60)
end
```

with the following output:

```
 ERhat(10):   1.0228e+00

------------------------------------------

 ERhat(20):   1.2080e+00

------------------------------------------

 ERhat(30):   1.4623e+00

------------------------------------------

 ERhat(40):   1.8228e+00

------------------------------------------

 ERhat(50):   2.3468e+00

------------------------------------------

 ERhat(60):   3.1121e+00
```

10.9 Irreducible Markov Chain with reward rates

In this example, the following language features are used: Markov chain specification with reward rates; user-defined functions; variable binding; built-in function exrss; built-in function tvalue, sum and sreward; evaluated names; and loop to print results. The following is the input to SHARPE:

```
markov irred readprobs
4 3 4*lambda
3 2 3*lambda
2 1 2*lambda
1 0 lambda
0 1 4*mu
1 2 3*mu
2 3 2*mu
3 4 mu
reward
```

```
4 40
3 25
2 10
1 3
end
4 1
end

bind lambda 100
bind mu 10
****ask for expected steady-state reward rate
expr exrss(irred)
****define a function giving transient reward rate
func treward(t) sum(i,0,4,sreward(irred,$i)*tvalue(t;irred,$i))

loop t,0,.05,.01
        expr treward(t)
end
end
```

which produces the following results:

```
 exrss(irred):   1.3005e+00

--------------------------------------------


t=0.000000
     treward(t):   4.0000e+01

t=0.010000
     treward(t):   9.1261e+00

t=0.020000
     treward(t):   3.2775e+00

t=0.030000
     treward(t):   1.8927e+00

t=0.040000
     treward(t):   1.4906e+00

t=0.050000
     treward(t):   1.3630e+00
```

10.10 Phase-type Markov Chain

In this example, the following language features are used: markov chain specification, cdf to print results, and imag on/off. The following is the input to SHARPE:

```
* a phase-type markov chain
* the only absorbing state is 6
markov phase
1 2 5
2 3 1
2 4 4
2 6 7
3 1 3
3 4 11
3 5 5
3 6 3
4 3 15
5 4 6
end
1 1
end

* cdf (phase) is a function of t giving
* the probability that absorption is
* reached in time <= t.
* cdf (phase) is the same as cdf(phase,6)

cdf(phase)

* print the result with i instead of
* sin/cos

imag on
cdf(phase)

* print transient probability function.
* this is a function of t giving the
* instantaneous probability of being in
* state 4 at time t

imag off
cdf(phase,4)
end
```

which results in the output of:

```
CDF for system phase:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.5053e-01 t(  0) exp(-1.2207e+00 t)
 + -7.1142e-01 t(  0) exp(-5.2558e+00 t)
 +  3.6014e-01 t(  0) exp(-1.1174e+01 t) cos  1.8397e+00 t
 + -2.4602e-01 t(  0) exp(-1.1174e+01 t) sin  1.8397e+00 t
 +  1.8126e-03 t(  0) exp(-3.1176e+01 t)

mean: 6.4035e-01
variance: 5.1045e-01

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

```
CDF for system phase:

    1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.5053e-01 t(  0) exp(-1.2207e+00 t)
 + -7.1142e-01 t(  0) exp(-5.2558e+00 t)
 +  1.8007e-01+1.2301e-01i t(  0) exp(-1.1174e+01+1.8397e+00i t)
 +  1.8007e-01-1.2301e-01i t(  0) exp(-1.1174e+01-1.8397e+00i t)
 +  1.8126e-03 t(  0) exp(-3.1176e+01 t)

mean: 6.4035e-01
variance: 5.1045e-01

-------------------------------------------

information about system phase node     4
transient state probability:

    1.9374e-01 t(  0) exp(-1.2207e+00 t)
 + -1.0958e-01 t(  0) exp(-5.2558e+00 t)
 + -9.4970e-02 t(  0) exp(-1.1174e+01 t) cos  1.8397e+00 t
 + -5.7818e-01 t(  0) exp(-1.1174e+01 t) sin  1.8397e+00 t
 +  1.0807e-02 t(  0) exp(-3.1176e+01 t)
```

10.11 Reliability block diagram

In this example, summarized below, the following language features are used: reliability block diagram specification, user-defined distributions, built-in function values, and cdf to print results.

The input to SHARPE:

```
* user-defined distributions for RELCOMP

poly activeE (lambda) gen\
  1,0,0\
 -2,0,-lambda\
  1,0,-2*lambda

poly activeU (lambda,s) gen\
  1,0,0\
 -1,0,-lambda\
 -1,0,-mu\
  1,0,-(lambda+mu)

poly standbyE (lambda,s) gen\
  1,0,0\
 -1,0,-lambda\
 -lambda,1,-(lambda+s)
```

```
poly standbyU (lambda,mu,s) gen\
  1,0,0\
 -1,0,-lambda\
 -lambda/(mu-lambda), 0, -(lambda+s)\
  lambda/(mu-lambda), 0, -(mu+s),

* note this distribution has mass only
* at zero and infinity

poly oneshot(p) gen\
 1-p,0,0
block KN (lambda, k, n)
comp x exp(lambda)
kofn top n-k+1,n,x
end

* here we define a distribution to be
* the CDF of a block diagram

poly binomial (lambda,k,n)\
cdf (KN;lambda,k,n)

block DP
comp receiver exp (.0002)
comp tuner activeE (.00025)
comp mux standbyE (.00025, .000005)
comp cpu binomial (.0004, 2, 3)
series DP receiver tuner mux cpu
end

* probability of failure at t = 20

expr 1-value(20;DP)

* CDF of time-to-failure

cdf(DP)
end
```

which produces the following result:

```
 1-value(20;DP) :   9.9578e-01

-------------------------------------------


CDF for system DP:

   -1.5000e-03 t(  1) exp(-1.5050e-03 t)
 +  7.5000e-04 t(  1) exp(-1.7550e-03 t)
 +  1.0000e-03 t(  1) exp(-1.9050e-03 t)
 + -5.0000e-04 t(  1) exp(-2.1550e-03 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -6.0000e+00 t(  0) exp(-1.5000e-03 t)
 +  3.0000e+00 t(  0) exp(-1.7500e-03 t)
 +  4.0000e+00 t(  0) exp(-1.9000e-03 t)
 + -2.0000e+00 t(  0) exp(-2.1500e-03 t)

mean: 1.3615e+03
variance: 9.9601e+05
```

10.12 Product-form queueing network

This is Example 9.6 from KST. In this example, the following language features are used: product-form queueing network specification; built-in functions tput, util, qlength, rtime; and loop to print results. The input to SHARPE is:

```
* central-server queueing system

bind
p1 0.667
p2 o.233
end

pfqn csm
cpu disk1 p1
cpu disk2 p2
disk1 cpu 1
disk2 cpu 1
end

* fcfs servers
cpu fcfs 1000/20
disk1 fcfs 1000/30
disk2 fcfs 1000/42.918
end

* number of jobs
chain1 custs
end

loop i,2,10,2
 bind custs i
 expr tput(csm,cpu)
 expr util(csm,cpu)
 expr qlength(csm,cpu)
 expr rtime(csm,cpu)
end


end
```

which produces the output:

```
i=2.000000
```

```
     custs <- 2.000000
     tput(csm,cpu):   2.9406e+01
     util(csm,cpu):   5.8811e-01
     qlength(csm,cpu):   8.2331e-01
     rtime(csm,cpu):   2.7998e-02

i=4.000000
     custs <- 4.000000
     tput(csm,cpu):   3.7976e+01
     util(csm,cpu):   7.5952e-01
     qlength(csm,cpu):   1.7202e+00
     rtime(csm,cpu):   4.5298e-02

i=6.000000
     custs <- 6.000000
     tput(csm,cpu):   4.1733e+01
     util(csm,cpu):   8.3465e-01
     qlength(csm,cpu):   2.6591e+00
     rtime(csm,cpu):   6.3717e-02

i=8.000000
     custs <- 8.000000
     tput(csm,cpu):   4.3753e+01
     util(csm,cpu):   8.7506e-01
     qlength(csm,cpu):   3.6209e+00
     rtime(csm,cpu):   8.2758e-02

i=10.000000
     custs <- 10.000000
     tput(csm,cpu):   4.4992e+01
     util(csm,cpu):   8.9983e-01
     qlength(csm,cpu):   4.5955e+00
     rtime(csm,cpu):   1.0214e-01
```

10.13 Generalized Stochastic Petri Net

In this example, which is summarized below, the following language features are used: generalized stochastic petri net specification; user-defined variables; built-in functions prempty, etok, tput, util; and expr to print results. The input to SHARPE:

```
bind
lambda 1.2
mu 2.0
gamma 0.0001
tau 0.1
K 10
end
* M/M/1/K queueing system where
* server can fail and be repaired

gspn mm1k-fail
* places and initial number of tokens
```

```
jobsource K
queue 0
serverup 1
serverdown 0
end
* timed transitions and rates
job-arrival ind lambda
service ind mu
failure ind gamma
repair ind tau
end
* immediate transitions (none here)
end
* enabling arc place -$>$
* and number of tokens
jobsource  job-arrival 1
queue service 1
serverup failure 1
serverdown repair 1
end
* transition -$>$ place and number of tokens
job-arrival queue 1
service jobsource 1
failure serverdown 1
repair serverup 1
end

* and number of tokens
serverdown service 1
end

* use variables to define some measures of interest

* probability that the server is idle
var Pidle prempty(mm1k-fail,queue)

* probability that a job is rejected - this happens
* if all K tokens are in the queue and none are in
* the jobsource place

var Preject prempty(mm1k-fail,jobsource)

* reject rate
var Lreject lambda * prempty(mm1k-fail,jobsource)

* average queue length is the average number
* of tokens in the queue place
var avquelenth etok(mm1k-fail,queue)

* throughput is the throughput of the server
* transition
var thruput tput(mm1k-fail,service)

* utilization is the utilization of the server
* transition

var utilization util(mm1k-fail,service)

expr Pidle
expr Lreject, Preject
expr avquelenth
expr thruput, utilization

end
```

produces the following results:

```
 Pidle :   4.0083e-01

--------------------------------------------

 Lreject:   3.6034e-03
 Preject :   3.0029e-03

--------------------------------------------

 avquelenth :   1.4688e+00

--------------------------------------------

 thruput:   1.1964e+00
 utilization :   5.9820e-01
```

10.14 Series-parallel acyclic graph

In this example, the following language features are used: series-parallel acyclic graph specification, user-defined distribution with mass at infinity, and cdf to print results. The input to SHARPE:

```
* The following  user-defined distribution
* is ``defective''. f is the probability
* that the task never completes.

poly F(f, u) gen 1-f, 0, 0 -(1-f), 0, -u

graph main (f1,f2,f3,f4,f5.f6,f7,f8,f9,f10,u)

e1 e2
e2 e3
e2 ez
e3 e5
e5 e7
e5 e9
e7 e10
```

```
e9 e10
ez e4
ez e6
e4 e8
e6 e8
e8 e10
end
exit e2 prob
exit e5 min
exit ez max
prob e2 e3 p23

dist ez zero
dist e1 F(f1, u)
dist e2 F(f2, u)
dist e3 F(f3, u)
dist e4 F(f4, u)
dist e5 F(f5, u)
dist e6 F(f6, u)
dist e8 F(f8, u)
dist e9 F(f9, u)
dist e10 F(f10, u)
end
bind p23 .6
* f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 u
cdf(main;.02,.03,.04,.05,.06,.07,.08,.09,.04,.09,3)
cdf(main; 0, 0, 0, 0, 0, 0, 0, 0, .2, 0, 3)
end
```

produces the following results:

```
CDF for  system main:

probability at infinity: 2.5493e-01

continuous probability: 7.4507e-01

  -1.0774e-01 t( 5) exp(-3.0000e+00 t)
+ -4.8496e+00 t( 4) exp(-3.0000e+00 t)
+ -2.3943e-01 t( 3) exp(-3.0000e+00 t)
+ -6.4662e+00 t( 2) exp(-3.0000e+00 t)
+ -1.5962e-01 t( 1) exp(-3.0000e+00 t)
+  7.4507e-01 t( 0) exp( 0.0000e+00 t)
+ -1.4369e+00 t( 0) exp(-3.0000e+00 t)
+  6.9186e-01 t( 0) exp(-6.0000e+00 t)

mean  and variance are conditional on finite time

mean: 1.8452e+00
variance: 5.9113e-01

------------------------------------

CDF for system main:
  -2.4300e-01 t( 5) exp(-3.0000e+00 t)
+ -6.3450e+00 t( 4) exp(-3.0000e+00 t)
+ -2.3943e-01 t( 3) exp(-3.0000e+00 t)
+ -8.4600e+00 t( 2) exp(-3.0000e+00 t)
+ -3.6000e-01 t( 1) exp(-3.0000e+00 t)
+  1.0000e+00 t( 0) exp( 0.0000e+00 t)
+ -1.8800e+00 t( 0) exp(-3.0000e+00 t)
+  8.8000e-01 t( 0) exp(-6.0000e+00 t)

mean: 1.8533e+00
variance:  5.9627e-01
```

10.15 Markov Chain inside a graph

In this example, the following language features are used: markov chain specification, series-parallel acyclic graph specification, and use of cdf as a component distribution specification. The input to SHARPE:

```
bind
r1 2
r2 5
r3 4
r4 3
r5 8
r6 3
r9 2
p63 .8
end

* Markov chain for task loop inside graph
markov LOOP
3 6 r3
6 3 r6*p63
6 d r6*(1-p63)
end
3 1
end

* Task graph with CDF of another model
* as a parameter
graph outer
1 2
1 3-6
2 4
2 5
4 9
5 9
3-6 9
end
exit 1 max
exit 2 max
dist 1 exp(r1)
dist 2 exp(r2)
dist 4 exp(r4)
dist 5 exp(r5)
dist 9 exp(r9)
dist 3-6 CDF(LOOP)
end
* cdf(outer) gives the exact cumulative
* distribution function for the time-to-finish
*of the graph with loop.
cdf(outer)
end
```

produces the following output:

```
CDF for system outer:

   -1.5832e+00 t(  1) exp(-2.0000e+00 t)
 +  1.0000e+00 t(  0) exp( 0.0000e+00 t)
 + -1.5758e+00 t(  0) exp(-3.6153e-01 t)
 +  4.8098e+00 t(  0) exp(-2.0000e+00 t)
 + -1.0000e+01 t(  0) exp(-3.0000e+00 t)
 +  5.7051e+00 t(  0) exp(-3.3615e+00 t)
 +  2.9630e-01 t(  0) exp(-5.0000e+00 t)
 + -2.4958e-01 t(  0) exp(-5.3615e+00 t)
 +  1.0708e-02 t(  0) exp(-6.6385e+00 t)
 +  1.8519e-01 t(  0) exp(-8.0000e+00 t)
 + -1.7422e-01 t(  0) exp(-8.3615e+00 t)
 + -9.8715e-03 t(  0) exp(-9.6385e+00 t)
 + -4.1152e-02 t(  0) exp(-1.1000e+01 t)
 +  4.0226e-02 t(  0) exp(-1.1362e+01 t)
 +  1.6533e-03 t(  0) exp(-1.1638e+01 t)
 +  2.4039e-03 t(  0) exp(-1.4638e+01 t)
 + -7.8503e-04 t(  0) exp(-1.7638e+01 t)

mean: 3.9701e+00
variance: 7.9428e+00
```

A Using sharpe

SHARPE is a powerful tool for solving a variety of mathematical models. We now step through the example of section 10.1 to illustrate the way of using SHARPE.

This example gives the solution for the Memory interference model (from section 7.5.1 on page 326, Probability and Statistics with Reliability Queuing and Computer Science Applications). The system is a shared memory multiprocessor system. The processors' ability to share the entire memory space provides a convenient means of sharing information and provides convenient means of sharing information and provides flexibility in memory allocation. The price of sharing is the contention for the shared resource. To reduce contention, the memory is usually split up into modules, which can be accessed independently and concurrently with other modules. When more than one processor attempts to access the same module, only one processor can be granted access, while other processors must await their turn in a queue. The effect of such contention, or interference, is to increase the average memory access time.

The model for this system is given on page 328. For clarity the program of section 10.1 is duplicated below:

```
markov mem_interfere
11 02 q2*q2
11 20 q1*q1
02 11 q1
20 11 q2
end

bind
q1 .6
q2 .4
end

expr prob(mem_interfere,11)
expr prob(mem_interfere,02)
expr prob(mem_interfere,20)

end
```

The first line of the program specifies the system type and the system name which the author of the program chooses to use. All the lines till an end is encountered are transition probabilities. The transition probabilities are specified from left to right. That is, the transition probability of going from state 11 to state 20 is q2*q2.

The next section binds the variable q1 and q2 to numerical values. These values have been chosen arbitrarily.

The last section uses the built-in functions expr and prob to find and print out the steady state probabilities of all the states. The output produced is exactly the same as in the right column of the figure in section 10.1. The last end, ends the model specification.

It is easily seen that the transition probabilities from a state do not sum to 1 despite it being a irreducible discrete-time chain. It is because SHARPE does not allow self loops. A discrete-time chain is specified as a continuous-time chain with the transition rates being replaced with the transition probabilities. The probabilities associated with the self loops are not entered. SHARPE by itself substitutes the values and solves the chain as if it were a continuous-time chain. This does not affect the solution of the discrete chain. You should try to prove it mathematically.

The name of the file containing the SHARPE program can have any name and no extension is required (you may put it in if you want).

Following are the step in using SHARPE:

1. Write the program for solving the model in SHARPE.
2. At the prompt type the command words in bold face as it is:
   sharpe *filename*

SHARPE will print the results on your screen. To store the output in a file the command

sharpe *filename1* > *filename2*

is used, where *filename1* contains the SHARPE program and *filename2* is the file to which you want to send the output. SHARPE takes options for running the programs on command line. For the options refer to the language description