PERFORMANCE AND RELIABILITY ANALYSIS OF COMPUTER SYSTEMS

An Example-Based Approach Using the SHARPE Software Package

PERFORMANCE AND RELIABILITY ANALYSIS OF COMPUTER SYSTEMS

An Example-Based Approach Using the SHARPE Software Package

Robin SAHNER
Urbana, IL

Kishor S. TRIVEDI
Duke University
Durham, N.C.

Antonio PULIAFITO
University of Catania
Catania, Italy

KLUWER ACADEMIC PUBLISHERS
Boston/London/Dordrecht

CONTENTS

PART I

APPENDICES

A

SHARPE COMMAND LINE SYNTAX

The SHARPE program has two modes of operation: it can read input from one or more files (batch mode) or from a terminal (interactive mode). The language used in interactive mode is a subset of the batch-mode language; some of the keywords are not used because the context of an interactive session makes them unnecessary. At the user's option, when in interactive mode SHARPE can copy the user's input into a file in the syntax suitable for batch-mode operation.

The SHARPE command line for batch mode is

sharpe [-v] [-nf] [-p<n|o>] [-e<1|2>] [-vo] [-s<c|u>] file [file ...]

The SHARPE command line for interactive mode is

sharpe [-v] [-nf] [-p<n|o>] [-e<1|2>] [-vo] [-s<c|u>] [-d datafile]

If there are no *file* arguments, SHARPE runs in interactive mode. If the **-d** flag is present, SHARPE runs interactively and saves the user's input in *datafile*. In interactive mode, SHARPE will prompt the user for input and read from the user's terminal. In general, if the user mistypes a line or types an invalid line, SHARPE will ignore the line and allow the user to try again. However, some errors are fatal. When SHARPE is saving a copy of the user's input in a file, it ignores lines containing errors.

If there are any *file* arguments, SHARPE runs in batch mode and reads its input from the files. When there is more than one input file, SHARPE does not care

where the file boundaries lie; it treats the files as if they were concatenated into one file.

In both batch and interactive mode, output appears on the user's terminal. The user may, of course, use an appropriate method (depending on the operating system) to redirect the output into a file.

The **-v** flag tells SHARPE to print verbose output. See Section B.5.5 for a list of extra information printed when this flag is present. Verbose mode can also be turned on and off while SHARPE is running using the statements **verbose on** and **verbose off**.

The **-nf** ("no force") flag has to do with the analysis of irreducible Markov chains, semi-Markov chains and GSPNs. Without this flag, SHARPE will switch automatically to an underrelaxation algorithm if its SOR iteration does not seem to be converging. With the flag, SHARPE asks the user to choose whether to stop, continue with SOR, or switch to relaxation with an ω parameter of the user's choice. See Sections **??**, **??** and B.7.2 for more details.

The **-pn**, **-po**, **-e1**, **-e2** and **-vo** flags choose between alternative algorithms provided for the analysis of phase-type Markov chains and GSPNs. For more information, see Appendix D.

The **-sc** and **-su** flags tell SHARPE whether the distributions assigned to the transitions in semi-Markov chains are to be **c**onditional or **u**nconditional. The default is conditional. The default can also be overridden on a per-chain basis in the statement that defines the chain name. See Sections **??** and B.4.2 for more information.

B

SHARPE LANGUAGE DESCRIPTION

B.1 CONVENTIONS

Keywords and necessary punctuation are given in **boldface**. Syntactic categories are given in *italics*. A line contained in angled brackets <> indicates an 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 ('|') indicate 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 ('\').

B.2 BASIC LANGUAGE COMPONENTS

B.2.1 Comments

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

B.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.

B.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.

B.2.4 Taking Input From Secondary Files

A line of the form

include *filename*

causes input to be taken from *filename*. This may appear anywhere in either the "primary" input file (the one given on the command line) or in files that are "included." There is no protection against infinitely recursive includes.

B.2.5 Names

A *name* consists of a letter followed by any combination of letters, digits, colons, pound signs ('#'), question marks, underscores and periods. *Names* are used for variables, function and exponential polynomial names, parameters, and indices in **sum** functions.

Names may be any length, but only the first part of each name is significant. Early versions of SHARPE looked at the first 14 characters; later versions provide the number of significant characters via the command **info constants**.

In the remainder of this document, identifiers ending in *_name* are assumed to be of type *name*.

B.2.6 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 name 4A-1B-5. For examples of the use of *evaluated words*, see Sections **??** and **??**.

Words may be any length, but only the first part is significant. Early versions of SHARPE looked at the first 14 characters; later versions provide the number of significant characters via the command **info constants**. In the case of *evaluated words*, the truncation occurs after evaluation.

In the remainder of this document, identifiers ending in *_word* are assumed to be of type *word* and identifiers ending in *_eword* are assumed to be of type *evaluated word*.

B.2.7 Arithmetic Expressions

Expressions (*expression*) are written in infix form. The following operators are allowed: addition('+'), subtraction ('−'), negation ('−'), division ('/'), multiplication, ('*'), exponentiation ('^') and "power of e" ('^'). 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. All operations are floating point.

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 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 B.6. 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 operands are *constants*.

B.2.8 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

func *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.

B.2.9 Scope of Names and Words

Simple variables, defined variables, function names, exponential polynomial 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 exponential polynomial definition. The index of a loop is local to the loop. The index in a sum function is local to the function.

B.2.10 Parameter and Argument Lists

A parameter list (*param_list*) has the form

name, name, ... , name

Parameter lists are used when in the definitions of functions, exponential polynomials and models. An argument list (*arg_list*) has the form

expression, expression, ... , 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.

B.3 SPECIFICATION OF EXPONENTIAL POLYNOMIAL FUNCTIONS

An exponential polynomial function (EP) is a finite exponential polynomial:

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

See Sections **??** and **??** for background information about exponential polynomials and mixture distributions. These are the built-in forms for specifying an EP (*ep*):

1. **zero**

This specifies the discrete function having all of its mass at zero.

2. **inf**

   This specifies the discrete function having all of its mass at ∞.

3. **prob** (p)

   This specifies a discrete function having mass p at zero and $1-p$ at ∞.

4. **exp** (λ)

   This specifies the exponential polynomial $F(t) = 1 - e^{-\lambda t}$.

5. **gen** *triple, triple, ...*

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

   $$a_j, k_j, b_j$$

   where a_j and b_j may be real or integer, $b_j < 0$, and k_j must be a non-negative integer.

6. **cgen** *five-tuple, five-tuple, ...*

   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 a non-negative integer, the rest of the numbers may be real or integer and $real(b_j) < 0$. The exponential polynomial 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.

7. **tgen** *n-tuple, n-tuple, ...*

   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 e real or integer; again $b_j < 0$.

8. *defined_ep (arg_list)*

   This specifies a user-defined exponential polynomial. The parentheses must be present even if there are no parameters.

9. **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 exponential polynomial is that printed by the **cdf** keyword. The number of arguments must match the number of parameters (possibly none) in the system definition.

To create a user-defined exponential polynomial, the following statement is used:

poly *name* (*param_list*) *ep*

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

B.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 in 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.

B.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.

Markov Chains with Absorbing States

A 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, nonabsorbing states must have nonzero reward rates. By default, a state that is not assigned a reward rate is assumed to have a reward rate of 0. If the **reward** keyword is followed by **default** *expression*, the default reward for this chain is changed to *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 0. The sum of all assigned initial probabilities must be 1.

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 1, and all other nodes have initial probability 0. If more than one node has no incoming transitions, this section must not be empty.

Irreducible Markov Chains

An irreducible Markov chain is specified in one of two ways, either with or without initial state probabilities. If only a steady-state analysis of an irreducible 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 irreducible chains and did not expect initial state probabilities to be specified for irreducible chains. For the sake of compatibility, SHARPE assumes that irreducible chains will be followed by initial state probabilities if and only if the line containing the chain name ends in the keyword **readprobs**. An 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 an irreducible chain has been specified without initial state probabilities, **tvalue** cannot be applied to it. When an irreducible chain has been specified with **readprobs**, either **prob** or **tvalue** can be applied to it. An 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

B.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 ep*>
* 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 all transition from state from A other than to B are disallowed.

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, $Q(t)$ is the unconditional probability that the time from entering A to entering B is less than or equal to t. The function $Q(t)$ is thus defective (less than 1 as $t \to \infty$) 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 unconditional.

An irreducible semi-Markov chain is specified as follows:

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

B.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:

1. **comp** *name ep*

   This is a basic component type. It is assigned a name and an exponential polynomial.

2. **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.

3. **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.

4. **kofn** *name1 expression, expression, name2*

   This represents a 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.

5. **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.

B.4.4 Fault Trees

A fault tree is specified by the following:

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

An *ftreeline* has one of the following forms:

1. **basic** *name ep*

   This is a basic event (or component) type. It is assigned a name and an exponential polynomial. Whenever this name appears later in the fault tree specification, it is interpreted as being a physically distinct copy of an event type having the assigned exponential polynomial.

2. **repeat** *name ep*

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

3. **transfer** *name name*

   The second name must have been previously defined using either **basic** or **repeat**. Whenever the first name appears later in the fault tree specification, it is interpreted as being the same physical event as the second name.

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

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

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

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

6. **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.

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

   This represents a k-out-of-n gate whose inputs need not be 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 the system only fails if k of the inputs fail.

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

B.4.5 Reliability Graphs

A reliability graph is specified by the following:

relgraph *name* { (*param_list*) }
* section 1: unidirectional edges
<*edge_name edge_name ep*>
* section 2: bidirectional edges (optional)

{ **bidirect**
<*edge_name edge_name ep*> }
end

In the first section, unidirectional edges are specified. A path exists from the first *edge_name* to the second *edge_name*. The *ep* is the exponential polynomial 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 *ep* as the time-to-failure CDF.

B.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_type 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 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. *station_type* is chosen from a pre-defined 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-serve 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, preemtive 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 has the same rate.

6. *station_name* **lds** *rate_or_loop*, *rate_or_loop*, ...

   There is one server, whose service rate depends on the number of jobs at the station. The keyword **lds** is followed by a series of rates or loops. A rate is any *expression*. A loop has the form

   loop (*index,low,high,increment,expression*)

   Each loop is expanded into a series of rates (see Section **??** for an example).

   After expansion of loops, 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 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, the chain must be given a name (which is never used).

B.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

The *expression number_of_servers* is present if and only if *station_type* is **ms** (multiple server). Multiple rates (and/or loops) are expected if and only if *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.

B.4.8 Generalized Stochastic Petri Nets

A generalized stochastic Petri net (GSPN) 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 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 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 many 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.

B.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:

1. **dist** *name ep*

   This assigns the given *ep* to the given graph node. An *ep* must be specified for each graph node.

2. **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 specified. 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 exponential polynomial 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 $E.g$.

3. **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 1. If one value is missing, the sum of the values must be less than 1 and SHARPE will compute the missing value.

4. **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

1. **prob**

The parallel subgraphs are probabilistic.

2. **max**

   All of the parallel subgraphs must complete before going on.

3. **min**

   One of the parallel subgraphs must complete before going on.

4. **kofn** *expression, expression*

   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 a node with **kofn** exit type have exactly n following parallel subgraphs.

B.5 ASKING FOR RESULTS

B.5.1 Printing Results of Model Analysis

SHARPE provides the following statements for printing the results of model analysis:

1. **cdf** *(system_name* {;*arg_list*}**)**
   cdf (*chain_name*{, *state_eword*} {;*arg_list*}**)**

   The keyword **cdf** asks for an exponential polynomial result and its mean and variance. The name **cdf** was chosen because the result is usually a cumulative distribution function (CDF). However, it may have another interpretation depending on the system and the meaning of the functions assigned to the system components. The first form is for graphs, block diagrams, fault trees, reliability graphs and non-irreducible GSPNs. The second form is for Markov and acyclic semi-Markov chains. The results are:

   (a) **series-parallel acyclic directed graph.** Generally the function assigned to each component is the CDF of its execution time; **cdf** for the graph is the graph execution time. If multiple-path information was requested, the CDF is given for each path.

(b) **reliability block diagram, fault tree, reliability graph.** If the function assigned to each component is the CDF of its failure time, **cdf** gives the system failure time CDF. If the function assigned to each component is the instantaneous or steady-state availability, **cdf** gives the instantaneous or steady-state system availability.

(c) **acyclic semi-Markov chains, acyclic and phase-type Markov chains.** If no state name is given, **cdf** gives the distribution function for the time until some absorbing state is reached. If the name of an absorbing state is given, **cdf** gives the distribution function for the time until the given state is reached, conditional on that state being reached, and also prints the probability that the state is reached. If the name of a transient state is reached, **cdf** gives the transient probability function of being in that state.

(d) **irreducible Markov chain.** A state name must be given; **cdf** gives the transient probability function of being in that state.

(e) **non-irreducible GSPN**: **cdf** gives the CDF for the time until reaching an absorbing marking.

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

   The keyword **lcdf** asks for the exponential polynomial function of t giving the conditional probability that the state has been left by time t, given that the state was visited at all. SHARPE also prints the mean and variance of the CDF and the probability of visiting the state.

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

   This statement is valid for fault trees and reliability graphs. The result of the system analysis is printed symbolically in terms of the functions assigned to 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 is the function assigned to component or edge i and P_i is $1 - Q_i$.

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

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

5. **fetok** (*system_name*,*place_eword* {;*arg_list*})
   fprempty (*system_name*,*place_eword* {;*arg_list*})
   futil (*system_name*,*transition_eword* {;*arg_list*})
   ftput (*system_name*,*transition_eword* {;*arg_list*})

These statements are valid for GSPNs. They are transient functions of a time variable t, given symbolically in the time variable. **fetok** gives the average number of tokens in the specified place. **fprempty** gives the probability that the specified place is empty. **futil** and **ftput** give the utilization and throughput, respectively, of the specified transition. For more ways of getting GSPN results, see Section B.6.6.

6. **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 functions names that take a time parameter (see Section B.6). 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.

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

   SHARPE prints the value of the expression(s).

B.5.2 Number of Digits Printed

The statement

format *constant_expression*

specifies the number of digits after the decimal point to be printed in results. This in no way changes the way calculations are carried out internally, which is however the machine implements the C **double** data type.

B.5.3 Format for Complex Numbers

The keyword **imag** is used to control whether a CDF containing imaginary 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.

B.5.4 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.

B.5.5 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
- 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 that are used (see Section B.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

B.5.6 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 B.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

Here *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.

B.6 BUILT-IN FUNCTIONS

SHARPE provides built-in functions (*built_in_function*) that return information about model specification, provide values resulting from model analysis and provide summation functionality.

B.6.1 Functions of Exponential Polynomials

The following functions extract a value from an exponential polynomial result. (See Section B.5.1 for a description of the model types that yield exponential polynomial results.) If the model is an irreducible Markov chain,

state_eword must be present. If the model is an acyclic or phase-type Markov chain, *state_eword* is optional. For other model types, *state_eword* cannot be present.

1. **value** (*t*; *system_name* {,*state_eword*} {;*arg_list*})
   This gives the value at t of the exponential polynomial.
2. **mean** (*system_name* { ,*state_eword* } { ;*arg_list* })
   This gives the mean of the exponential polynomial.
3. **variance** (*system_name* { ,*state_eword* } { ;*arg_list* })
   This gives the variance of the exponential polynomial.
4. **pzero** (*system_name* { , *state_eword* } { ;*arg_list* })
   This gives the value at $t = 0$ of the exponential polynomial.
5. **pinf** (*system_name* {,*state_eword*} {;*arg_list*})
   This gives the limit at ∞ of the exponential polynomial.
6. **pcont** (*system_name* {,*state_eword*} {;*arg_list*})
   This gives the "continuous part" of the exponential polynomial. For an exponential polynomial $F(t)$, **pcont** gives $1 - F(0) - lim_{t\to\infty}F(t)$.

B.6.2 The function tvalue

The built-in function **tvalue** provides transient results for a single value of t obtained using algorithms that do not produce exponential polynomial functions. This contrasts with the function **value**, which tells SHARPE to produce an exponential polynomial function and then evaluate it at t. The function **tvalue** is valid with fault trees, block diagrams, reliability graphs, and Markov chains. The syntax is:

tvalue (*t*; *system_name* {,*state_eword*} {;*arg_list*})

B.6.3 The function prob

The built-in function **prob** provides state probabilities for Markov and semi-Markov chains. The syntax is:

prob (*system_name* , *state_eword* { ; *arg_list* }

For an acyclic semi-Markov chain or an acyclic or phase-type Markov chain, **prob** gives the probability that the given state was ever visited. For an irreducible Markov or semi-Markov chain, **prob** gives the steady-state probability for the given state.

B.6.4 Functions for Markov and semi-Markov Reward Models

The following built-in functions are specific to Markov and semi-Markov reward models:

1. **sreward** (*system_name*, *state_eword* {;*arg_list*})

   This gives the reward rate assigned to the given state.

2. **exrss** (*system_name* {;*arg_list*})

   This gives the expected steady state reward rate.

3. **exrt** (*t*; *system_name* {;*arg_list*})

   This gives the expected reward rate at time *t*.

4. **cexrt** (*t*; *system_name* {;*arg_list*})

   This gives the cumulative expected reward over (0,*t*).

5. **rvalue** (*r; system_name* {;*arg_list*})

   This applies to acyclic semi-Markov and acyclic and phase-type Markov models. It gives the probability that the accumulated reward is less than *r* when an absorbing state is reached.

B.6.5 Functions for Product-Form Queueing Networks

The following built-in functions are specific to single-chain and multi-chain product-form queueing networks.

1. **tput** (*system_name*,*eword* {;*arg_list*})

   This gives the throughput for a single-chain PFQN station.

2. **rtime** (*system_name*,*station_eword* {;*arg_list*})

   This gives the average response time of a single-chain PFQN station.

3. **qlength** (*system_name*,*station_eword* {;*arg_list*})

   This gives the average queue length at a single-chain PFQN station.

4. **util** (*system_name*,*eword* {;*arg_list*})

   This gives the utilization for a single-chain PFQN station.

5. **mtput** (*system_name*,*station_eword* {,*chain_eword*} {;*arg_list*})

   This gives the throughput for a multi-chain PFQN station, for a particular chain or sum over all chains.

6. **mrtime** (*system_name*,*station_eword* {,*chain_eword*} {;*arg_list*})

   This gives the average response time for a multi-chain PFQN station, for a particular chain or sum over all chains.

7. **mqlength** (*system_name*,*station_eword* {,*chain_eword*} {;*arg_list*})

   This gives the average queue length at a multi-chain PFQN station, for a particular chain or sum over all chains.

8. **mutil** (*system_name*,*station_eword* {,*chain_eword*} {;*arg_list*})

   This gives the utilization for a multi-chain PFQN station, for a particular chain or sum over all chains.

B.6.6 Functions for GSPNs

The following built-in functions are specific to GSPNs. For more ways of getting results about GSPNs, see Section B.5.1.

steady-state measures

These functions describe the steady-state condition of a GSPN. For an acyclic or phase-type GSPN, this is the same as the condition of the GSPN when it has reached an absorbing marking.

- **etok** (*system_name*,*place_eword* {;*arg_list*})
  This gives the steady-state average number of tokens in the given place.
- **prempty** (*system_name*,*place_eword* {;*arg_list*})
  This gives the steady-state probability that the given place is empty.
- **util** (*system_name*,*transition_eword* {;*arg_list*})
  This gives the steady-state utilization for a transition.
- **tput** (*system_name*,*transition_eword* {;*arg_list*})
  This gives the steady-state throughput for a transition.

measures at time t, computed from time-symbolic result

These functions describe the condition of a GSPN at a specified time t. The function is computed by producing a time-symbolic exponential polynomial form for the desired function, then evaluating the exponomial at time t. Values for additional values of t are computed from the same exponential polynomial, as long as the model parameters do not change.

- **etokt** (*t*;*system_name*,*place_eword* {;*arg_list*})
  This gives the expected number of tokens in the given place at time t.
- **premptyt** (*t*;*system_name*,*place_eword* {;*arg_list*})
  This gives the probability that the given place is empty at time t.
- **utilt** (*t*;*system_name*,*transition_eword* {;*arg_list*})
  This gives the utilization of the given transition at time t.
- **tputt** (*t*;*system_name*,*transition_eword* {;*arg_list*})
  This gives the throughput of the given transition at time t.

measures at time t, computed numerically

These functions describe the condition of a GSPN at a specified time t. The function is computed numerically using uniformization. This is a more stable computation then the one used to produce an exponential polynomial result, but must be done separately for each value of t.

- **etoku** (*t*;*system_name*,*place_eword* {;*arg_list*})

  This gives the expected number of tokens in the given place at time *t*.

- **premptyu** (*t*;*system_name*,*place_eword* {;*arg_list*})

  This gives the probability that the given place is empty at time *t*.

- **utilu** (*t*;*system_name*,*transition_eword* {;*arg_list*})

  This gives the utilization of the given transition at time *t*.

- **tputu** (*t*;*system_name*,*transition_eword* {;*arg_list*})

  This gives the throughput of the given transition at time *t*.

time-averaged measurements

- **tavetokt** (*t*;*system_name*,*place_eword* {;*arg_list*})

  This gives the time-averaged number of tokens in the given place during $(0,t)$.

- **tavtputt** (*t*;*system_name*,*transition_eword* {;*arg_list*})

  This gives the time-averaged throughput of the given transition during $(0,t)$.

B.6.7 The built-in function sum

The built-in function **sum** provides the ability to take the summation of expressions. The syntax is;

sum (*index, low, high, expression*)

B.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.

B.7.1 Phase-type Markov Chain Analysis

SHARPE contains two algorithms for finding the CDF for the time to absorption in a phase-type Markov chain. These algorithms are used when the **cdf** or **value** keywords are used. Both algorithms allow the user to choose between two alternative implementations for two of the steps. Appendix D describes the alternatives and how to choose among them using command-line flags and SHARPE statements. If SHARPE is run with verbose output turned on, it prints the choices.

In addition to the algorithms for finding the CDF, SHARPE contains a "uniformization" algorithm that finds the probability that an absorbing state has been reached by a particular value of t. This algorithm is used when the built-in function **tvalue** is used. The uniformization algorithm is much more stable but it can take a lot longer than the CDF algorithms, especially for large values of t.

B.7.2 Analysis of Irreducible Markov and Semi-Markov Chains and GSPNs

To analyze irreducible Markov and semi-Markov chains, including the Markov chains to which irreducible GSPNs are reduced, SHARPE uses an iterative method called Successive Overrelaxation (SOR). This method is briefly described in Section **??**.

The iteration is considered to have converged when the "tolerance" is small enough, where the tolerance is the ratio between the largest difference between the same elements in the two most recent consecutive values for the steady-state probability vector (π in Equation **??**) and the largest (in absolute value) element in the most recent probability vector.

An important aspect of the SOR algorithm is the choice of ω in Equation **??**. SHARPE begins its analysis by setting $\omega = 1$. Periodically, SHARPE adjusts ω to try to speed up the convergence.

If the iteration does not converge to a solution after a certain number of iterations, the default behavior is to set $\omega = 0.9$ and continue. This is a switch to underrelaxation, which seems to have very good convergence properties over a wide range of examples.

SHARPE provides users more control over the choice of ω through the **-nf** flag on the command line. If **-nf** is used, then if SOR has not converged after some number of iterations, SHARPE prints the number of iterations and the tolerance and asks whether to stop, continue with its own value for ω, or use a user-supplied value for ω. As long as the user does not choose to stop, SHARPE will continue to prompt for a decision every time a certain number of iterations has gone by without convergence.

The number of iterations between user prompts depends on the number of states in the chain. The number of states also determines the maximum number of iterations SHARPE will do when the **-nf** flag is not present; this is much larger than the number of iterations between prompts.

The **tvalue** built-in function can 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.

B.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*

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 (see Sections B.7.1 and B.7.2).

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.

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

B.8 PROGRAM CONSTANTS

The following statement asks SHARPE to print information about the program constants it contains:

info constants

Here is a list of the quantities that are limited:

- length of an input line
- number of intervals in an **eval** statement

B.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):

echo *anytext*
include *filename*
var *name expression*
func *func_name* (*param_list*) *expression*
poly *name* (*param_list*) *ep*
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**]
voronoi [**on** | **off**]
loop *low*, *high*{,*increment*}
eigen1
eigen2
phold
phnew
type *system_name*
info [**epsilons** | **constants**]

C

USING SHARPE INTERACTIVELY

SHARPE syntax for interactive sessions is pretty much the same as for file input. The difference is that when used interactively, SHARPE expects a null line to end most constructs instead of the keyword **end**. During an interactive session, SHARPE allows the user to recover from input errors when it can.

Figure C.1 (in many pieces) shows an interactive SHARPE session. During the session, the block diagram of Figure ?? (Section ??) is specified and its failure-time distribution is evaluated over the interval (.5, 1.5) at intervals of 0.5.

Text in boldface is SHARPE output: prompts, error messages and results. Text in normal font is user input. The line numbers are for reference only.

```
 1  $ sharpe
 2
 3  Enter a line with one of the following forms:
 4     var       name expression
 5     func      name(params) expression
 6     poly      name(params) distribution
 7
 8     graph     name(params)
 9     markov    name(params) [readprobs]
10     semimark name(params) [cond|uncond]
11     block     name(params)
12     relgraph name(params)
13     ftree     name(params)
14     pfqn      name(params)
15     mpfqn     name(params)
16     gspn      name(params)
17
18     bind
19     format p
20
21     cdf      (name<,node><;args>)
22     reward   (name<,node><;args>)
23     eval (name<,node><;args>) low high increment \
24        <cdf|reward|exrt|cexrt>
25     expr  expression
```

Figure C.1 An interactive SHARPE session.

line 1: In response to the system prompt ($), the user types the command to run SHARPE interactively.

lines 2 to 25: SHARPE prints all of the available input line formats. This long prompt is only printed once, at the beginning of the session, unless the user asks to see it again (using **help**). Lines 23 and 24 will appear as one line, but it was broken up here to fit on the page.

```
26
27  block main
28
29  Enter lines of the form
30     comp       name distribution
31     parallel   name x1 x2 ...
32     series     name x1 x2 ...
33     kofn       name k, n, x1 x2 ...
34  End with a null line
35
36  comp proc
37  error: missing distribution type. Try again.
38
39  Distributions types are:
40        exp (rate)
41        dist_name (args)
42        zero
43        inf
44        gen
45        cdf (system<,node><;args>)
```

Figure C.1 *(continued)* An interactive SHARPE session.

line 27: the user asks to define a reliability block diagram called *main*.

lines 29 to 34: SHARPE prints the format for specifying a reliability block diagram.

line 36: the user enters a block diagram component but forgets to enter the failure-time distribution.

lines 37 to 45: SHARPE advises the user of the error, prompts for another try and reminds the user how to specify a distribution.

```
46  comp proc exp(plam)
47  comp mem exp(mlam)
48  parallel p2 proc proc
49  parallel m3 mem mem mem
50  series top
51  error: not enough components. Try again.
52  series top p2 m3
53
54
55  Enter another command line
56  Enter <help> to see the possibilities
57  Enter a null line if there are no more commands
58
59  bind
60
61
62  Enter lines of the form
63     symbol expression
64  End with a null line.
65  plam .00139
66
67
```

Figure C.1 *(continued)* An interactive SHARPE session.

lines 46 to 50: The user enters block diagram components and structures. On line 50, the user has forgotten to enter the substructures of the series structure.

line 51: SHARPE prints an error message and prompts for a retry

lines 52 and 53: the user correctly completes the block diagram specification, including a null line (line 53).

lines 55 to 57: SHARPE prompts for more input. Note that the list of input formats is not printed again, but the user may type **help** to see it.

line 59: The user asks to bind the variables to values.

lines 62 to 64: SHARPE gives instructions for binding.

lines 65 and 66: The user binds the variable *plam* and ends with a null line.

```
68  Enter another command line
69  Enter <help> to see the possibilities
70  Enter a null line if there are no more commands
71
72  cdf(main)
73  Enter a value for mlam: .00764
74
75  CDF for system main:
76
77      1.0000e+00 t( 0) exp( 0.0000e+00 t)
78  + -6.0000e+00 t( 0) exp(-9.0300e-03 t)
79  +  3.0000e+00 t( 0) exp(-1.0420e-02 t)
80  +  6.0000e+00 t( 0) exp(-1.6670e-02 t)
81  + -3.0000e+00 t( 0) exp(-1.8060e-02 t)
82  + -2.0000e+00 t( 0) exp(-2.4310e-02 t)
83  +  1.0000e+00 t( 0) exp(-2.5700e-02 t)
84
85  mean: 2.2609e+02
86  variance: 1.9742e+04
87
```

Figure C.1 *(continued)* An interactive SHARPE session.

lines 68 to 70: SHARPE prompts for more input.

line 72: The user asks to see the CDF for the system called *main*.

line 73: While SHARPE is analyzing the block diagram, it realizes that the variable *mlam* was never assigned a value. It prompts the user for a value for the variable. The value .007645 is typed by the user.

lines 75 to 86: SHARPE prints the system CDF and its mean and variance

```
 88  Enter another command line
 89  Enter <help> to see the possibilities
 90  Enter a null line if there are no more commands
 91
 92  eval (main) 5 1.5 .5
 93  warning: lower limit is greater than upper limit.
 94
 95  Enter another command line
 96  Enter <help> to see the possibilities
 97  Enter a null line if there are no more commands
 98
 99  eval (main) .5 1.5 .5
100       system main
101       t          F(t)
102
103  5.0000 e-01  5.3811 e-07
104  1.0000 e+00  2.3703 e-06
105  1.5000 e+00  5.8176 e-06
106
107  Enter another command line
108  Enter <help> to see the possibilities
109  Enter a null line if there are no more commands
110
111
112  $
```

Figure C.1 *(continued)* An interactive SHARPE session.

lines 88 to 90: SHARPE prompts for more input.

line 92: The user asks to have the CDF for system *main* evaluated. The user meant to type .5 for the lower bound, but left out the decimal point.

lines 92 to 97: SHARPE warns the user that the interval of evaluation is empty and prompts for another command.

line 99: The user types the line correctly.

lines 100 to 105: SHARPE prints the requested information and prompts for further input.

line 110: The user types a null line to end the session.

D

ALGORITHM CHOICES FOR PHASE-TYPE MARKOV CHAINS

SHARPE provides an "old" and a "new" algorithm for finding the CDF of the time to absorption in phase-type Markov chains. The "old" algorithm is called that because it was implemented first. The "new" algorithm is more efficient than the "old," ($\mathcal{O}(n^3)$ for the "new" as opposed to $\mathcal{O}(n^5)$ for the "old") but its numerical behavior is sometimes not as good. The "old" algorithm is the default.

Both the "old" and the "new" algorithms make use of an eigenvalue-finding algorithm. SHARPE contains two of these, the "first" and "second." The two algorithms are about as efficient as each other; each has better numerical behavior in some situations. The "first" eigenvalue-finder can only be used with the "old" algorithm; the "second" can be used with either algorithm.

Both the "old" and the "new" algorithms contain a step that consists of choosing a set of complex values for a variable. SHARPE contains two algorithms for doing this, the "simple" algorithm and the "voronoi" algorithm. Each leads to better numerical behavior in some situations. The "voronoi" value-chooser can be used only with the "new" algorithm; the "simple" can be used with either "old" or "new" algorithm.

The default is to use the "old" algorithm with the "first" eigenvalue-finder and the "simple" value-chooser. The **-pn** and **-po** flags tell SHARPE to use the "new" and "old" algorithms, respectively. The same thing can be done while SHARPE is running using the statements **phnew** and **phold**. The **-e1** and **-e2** flags tell SHARPE to use the "first" and "second" eigenvalue-finders, respectively. The same thing can be done while SHARPE is running using the statements **eigen1** and **eigen2**. The **-pn** flag implies **-e2** even if **-e2** is not

algorithm	eigenvalue finder	value chooser	statements	flags
slow	first	simple	(**phold**) (**eigen1**) (**voronoi off**)	(**-po**) (**-e1**) (no flag)
slow	second	simple	(**phold**) **eigen2** (**voronoi off**)	(**-po**) **-e2** (no flag)
fast	second	simple	**phnew** (**eigen2**) (**voronoi off**)	**-pn** **-e2** (no flag)
fast	second	voronoi	**phnew** (**eigen2**) **voronoi on**	**-pn** **-e2** **-vo**

Table D.1 Available phase-type analysis choices.

present on the command line. The **-vo** flag tells SHARPE to use the "voronoi" value-chooser. Use of "voronoi" can be turned on and off while SHARPE is running using the statements **voronoi on** and **voronoi off**.

Table D.1 summarizes the possible combinations of algorithms and methods 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 table, it means that the statement can be omitted because it is the default.