THE RECONSTRUCTION OF SHARPE

by

Hu Pan

Department of Computer Science
Duke University

Date: ______________________________

Approved:

____
Dr. Kishor S. Trivedi, Supervisor

____
Dr. Alvin R. Lebeck

____
Dr. Robert A. Wagner

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the Department of Computer Science
in the Graduate School of
Duke University

February 14, 2001

Contents

Chapter 1

Introduction

1.1 Symbolic Hierarchical Automated Reliability and Performance Evaluator(SHARPE)

Today's computer system design has become more and more complicated, so it is hard to predict the reliability, availability and serviceability characteristics of the resulting system. Also, it is too expensive and time-consuming to build even one prototype to take measurements. Even when that is not the case, if the model is a good match for the system, designers can more easily and quickly carry out trade-off studies, and compare design alternatives.

Generally, there are two kinds of models, discrete-event simulation models and analytic models, to help designers predict system behavior without having to build and measure a real system. For discrete-event simulation models, designers build a program to reproduce the running behavior of the modeled system and take measures of the behavior. On the other hand, for analytic models, designers use a set of formulas or equations to describe the system. By solving these equations, designers get the measures of the system. Although discrete-event simulation models provide more details of the system behavior, they consume more time and more computer resources than analytic models. The situation may become worse when designers want to vary many of the parameters of the system for many times. Analytic models are better abstractions of systems. But analysts have to be very careful on how to abstract these real-world systems.

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a software tool that analyzes a specific class of analytic models – stochastic models. It accepts a specification language, called SHARPE language, for building single or hierarchical combinations of analytic models and for choosing proper algorithms for analyzing them. Originally, SHARPE provided analysis algorithms for the following model types:

- Reliability block diagrams
- Fault trees
- Reliability graphs
- Series-parallel acyclic directed graphs
- Single-chain and multiple-chain product-form queueing networks
- Markov and semi-Markov chains
- Generalize Stochastic Petri nets

SHARPE language gives users the power to choose models that are a proper match of the problem under investigation and it is up to users to interpret the parameters of the system and the results of measurements in a meaningful way. So, users can freely deploy all the above models on any systems if necessary. In the SHARPE test-bed, different system examples, such as multiprocessor system, wireless system, software system, and token ring system, etc., are included. Another big plus for SHARPE is that it supports hierarchical modeling, which can solve very complicated systems without causing stiffness or largeness.

Programming of SHARPE began in the early 1980s, in C language. The first version appeared at 1986. At that time, computer world was still lacking the ideas of compiling tools such as lex and yacc. As time passed, more and more models have been added into

SHARPE which has gradually made the code, especially the language parsing part, difficult to manage. It has become more and more difficult to add new model types into SHARPE or to extend the SHARPE language syntax. So, **flex** – an advanced version of lex, and **Bison** – an advanced version of yacc have been used to reconstruct SHARPE. The C language compiler used is **GCC**. Introduction to **flex**, **Bison** and **GCC** is given at the section 1.2. Details of work that has been done in this project are listed at the section 1.3.

1.2 Tools used

1.2.1 GCC

GCC stands for ”GNU Compiler Collection”, where **GNU** was chosen following a hacker tradition, as a recursive acronym for ”GNU’s Not Unix”. **GCC** can compile programs written in C, C++, Objective C, Fortran, Java and CHILL. The main goal of **GCC** was to provide a good, fast compiler for computer platforms in the class that the GNU system aims to run on: 32-bit machines with 8-bit addresses bytes and several general registers, include AIX, DOS, HP-UX, SCO OpenServer/Unixware, Solaris (SPARC, Intel), SGI, and Windows 95, 98, NT, 2000. So, having been compiled successfully by **GCC**, SHARPE can easily be deployed on those popular platforms.

1.2.2 flex

flex, also from GNU, is a tool for generating *lexical scanners*, which are programs for recognizing lexical patterns in text. At first, **flex** reads a description of a lexical scanner from the given input files, or its standard input if no file names are given. The description is in the form of pairs of regular expressions and C code, called *rules*. According to the description, **flex** generates a C source file, ’**lex.yy.c**’, which defines a routine ’**yylex()**’. This file should be compiled and linked with the '**-lfl**' library to produce an executable. When the executable is running, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

The **flex** input file consists of four sections, separated by a line with just '%%' in it:

%{

C declarations

%}

definitions

%%

rules

%%

Additional C code

The *C declarations* section may define types and variables used in the actions. One can also use preprocessor commands to define macros, and use **#include** to include header files that do any of these things.

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start* conditions, which supports conditionally activating rules.

The *rules* section of the **flex** input contains a series of rules of the form:

pattern *action*

where the pattern must be un-indented, which is written using an extended set of regular expressions, and the action must begin on the same line, which can be any arbitrary C statement.

The *additional C code* section can contain any C code one wants to use.

The reason to choose **flex** rather than **lex** is that **lex** cannot handle languages, such as SHARPE language, having too many tokens.

1.2.3 Bison

Bison, as a GNU tool, is a general-purpose parser generator that converts a grammar description for an **LALR** context-free grammar into a C program to parse that grammar. It is upward compatible with **Yacc**: all properly-written **Yacc** grammars ought to work with Bison without change. **Bison** reads a **Bison** grammar file as input. The output is a C source file defining a function named **yyparse**, and the file is called a **Bison** parser. The job of the **Bison** parser is to group tokens into sets according to the grammar rules – for example, to group identifiers and operations into expressions. when it does this, it runs the actions for the grammar rules. The tokens come from a function called the *lexical scanner*, which, in this project, is the function **yylex** generated by **flex**.

The general form of a **Bison** grammar file is as follows:

%{

C declarations

%}

Bison declarations

%%

Grammar rules

%%

Additional C code

The *C declarations* may define types and variables used in the rules' actions. You can also use preprocessor commands to define macros used there, and use **#include** to include header files that do any of these things.

The *Bison declarations* declare the names of the terminal and non-terminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The *grammar rules* define how to construct each non-terminal symbol from its parts. The following rule defines a non-terminal *line* as newline character:

```
line : '\n'
;
```

The *additional C code* can contain any C code one wants to use.

1.3 Work of reconstruction

The programming of SHARPE began in the early 1980s, in C language. First version was released at 1986. At that time, computer world was still lacking of the ideas of compiling tools such as lex and yacc. As time passed, more and more models have been added into SHARPE which has gradually made the code, especially the language parsing part, difficult to manage. It has become more and more difficult to add new model types into SHARPE. So, **flex** – an advanced version of lex, and **Bison** – an advanced version of yacc have been used to reconstruct SHARPE (See Figure 1.1). Of course, the new version of SHARPE, which is backward compatible to the old version, supports old language syntax and all model types listed in section 1.1, Phased-mission systems, Multi-state fault trees, and repeated edges in reliability graphs from Xinyu Zang's work [18], Markov regenerative process from Wei Xie's work [17], and Stochastic Reward Nets, which is implemented by me. There is also fast Mean Time To Failure(MTTF) algorithm for Markov chains and semi-Markov chains [6], which is implemented by Wei Xie. All new changes to SHARPE are represented by rectangles with thick lines in Figure 1.1.

The only exception is the definition of a *name*. Now only any number of letters, digits, underline, and colon are used to define a *name*. *Names* can be any length, but SHARPE **only looks** at *the first 29 characters*, beyond that, SHARPE will ignore and provide a warning message to users.

Another extension to SHARPE language syntax is that *numbers* can be represented in scientific format so that 0.1 can be written as $1.0E-1$, which can ease the burden on users when coding their SHARPE input files.

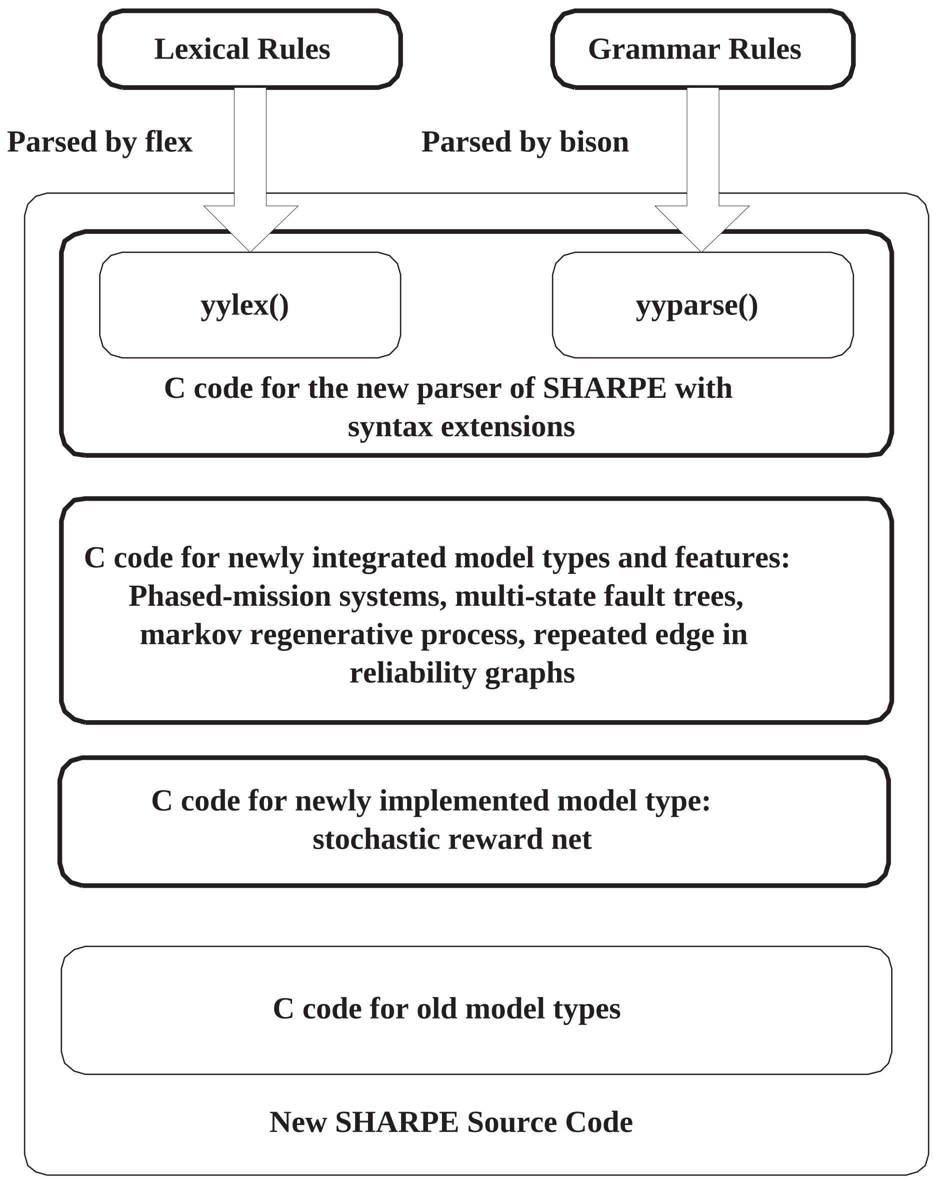


Figure 1.1: New SHARPE Construct

Other extensions to SHARPE language syntax will be mentioned when specific model types are introduced in subsequent chapters.

The new version of SHARPE accepts the following model types:

- Reliability block diagrams
- Fault trees
- Phased-mission systems
- Multi-state fault trees
- Reliability graphs with possibly repeated edges
- Series-parallel acyclic directed graphs
- Single-chain and multiple-chain product-form queueing networks
- Markov and semi-Markov chains
- Markov Regenerative Process
- Generalize Stochastic Petri nets
- Stochastic Reward Nets

There is also a test-bed which contains 41 directories and 978 test cases. The correctness of the new version of SHARPE is based on these test cases.

1.4 Scope of the thesis

The remainder of this thesis is organized into 2 chapters, as follows. Chapter 2 introduces how Stochastic Reward Nets (SRNs) has been implemented in SHARPE. Chapter 3 introduces all the model types which have been integrated into the new version of SHARPE. Examples have been selected to excise the features introduced. Appendix A includes all important data structures in SHARPE. Appendix B includes a partial SHARPE GUI document. Appendix C includes extra examples referenced in this thesis.

Chapter 2

New Model Type in SHARPE – Stochastic Reward Nets (SRNs)

2.1 Background

2.1.1 Petri Nets (PNs) and Generalized Stochastic Petri Nets (GSPNs)

Petri nets (PNs) were introduced by C.A. Petri in 1962 [12]. As a a bipartite directed graph, a PN consists two types of nodes: *places*, P, and *transitions*, T. Its directed arcs fall in two categories: *input arcs*, which lead from an *input place* to a transition, and *output arcs*, which connect a transition to an *output place*. Arcs cannot connect the same type of nodes, such as from places to places or from transitions to transitions. A non-negative number of *token*s can be assigned to each place. A *marking* $m \in \mathcal{M}$ is defined as a possible distribution of tokens to all places in the PN. Let P denotes the set of places. Then a marking m represents a multi-set, $m \in \mathcal{M} \subset \mathcal{IN}^{|\mathcal{P}|}$, describing the number of tokens in each place. See Figure 2.1. We use circle to denote a place, and a rectangle or a bar to denote a transition. Places represent conditions in the system being modeled. Transitions represent events occurring in the system. *Input arcs* are directed arcs from places to transitions representing the requirement or conditions for the event, which is denoted by the transition, to be triggered; *output arcs* are directed arcs from transitions to places representing the state or condition resulting from the occurrence of an event; *input places* of a transition are the set of places that are connected to the transition through input arcs; *output places* of a transition are the set of places to which output arcs exist from the transition.

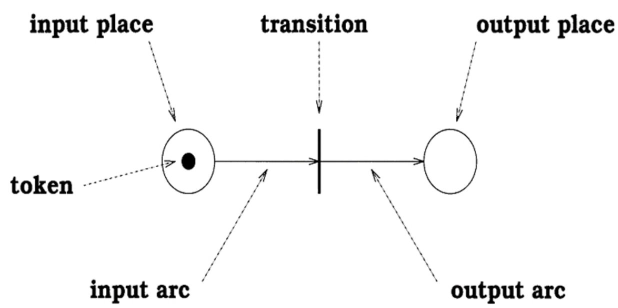


Figure 2.1: Basic components of a Petri net

A transition is *enabled* in the PN if the conditions for the corresponding event are met, which means all of the transition's input places contain at least one token. A transition is always enabled if there is no input arc connected to it. In the situation when more than one transitions is *enabled*, *priority* may be introduced to resolve the conflict (see Chapter 2.1.2). When an enabled transition *fire*s, one token from each input place is removed and one token is added to each output place (See Figure 2.2). The firing of a transition may transform a PN from one marking into another, changing the state or condition. *Marking* of a Petri net is the distribution of tokens among the places of the net. Given an *initial* marking, the *reachability set*, $\mathcal{RS}$, is defined as the set of markings reachable through any firing sequences of transitions beginning from the initial marking (See Figure 2.2). A *reachability graph* is represented as a directed graph with markings as its nodes and marking-to-marking transitions as its directed arcs. Depending on the situation, a $\mathcal{RS}$ could be infinite. Markings in which no transition is enabled are called *absorbing* markings.

Arcs of PNs can be extended to define *arc cardinality* or *multiplicity*. A transition is *enabled* when each input place connected to it contains at least as many tokens as the cardinality of the input arc. When the transition fires, the number of tokens removed from

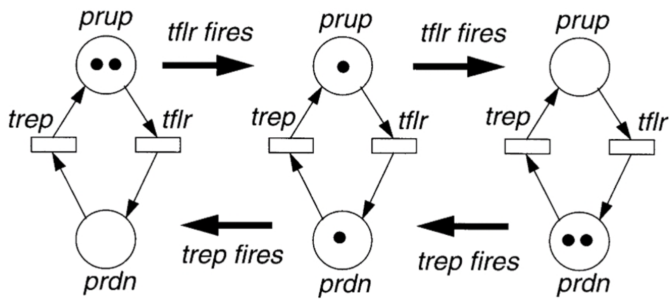


Figure 2.2: Enabling and Firing of Transitions

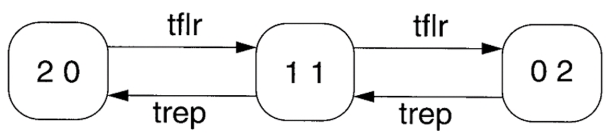


Figure 2.3: Reachability Set

the input place is the cardinality of the corresponding input arc, and the number of tokens added into the output place is the cardinality of the corresponding output arc (See Figure 2.4).

Further, *inhibitor arc*s are introduced as the third category of PN arcs. An inhibitor arc is drawn from a place to transition. The place is called *inhibitor place*. *Inhibitor arc* inhibits the firing of a transition when the corresponding inhibitor place has at least as many tokens as the cardinality of the corresponding inhibitor arc, even under the situation that all other conditions for enabling the transition are met. *Inhibitor arc*s are also directed arcs with a small circle rather than an arrow-head showing its direction (See Figure 2.4).

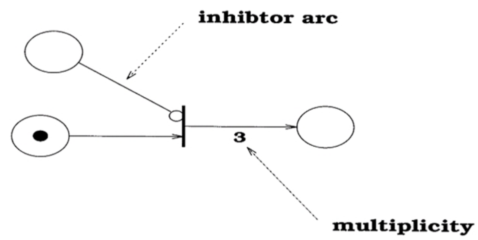


Figure 2.4: Extension of GSPN 1

Another way of extending PNs is to assign time with the firing of transitions, resulting in timed Petri nets. Generalized Stochastic Petri Nets (GSPNs) are one of them. In GSPNs, there are two types of transitions: *timed* transitions whose firing time is exponentially distributed and *immediate* transitions whose firing time is constant zero. *Timed* transitions are denoted by empty rectangles, while *immediate* transitions are drawn as bars.

The markings in the reachability set $\mathcal{RS}$ of a GSPN are partitioned into two sets: the *vanishing* markings $\mathcal{V}$ and the *tangible* marking $\mathcal{T}$. So, $\mathcal{M} = \mathcal{V} \bigcup \mathcal{T}$. Vanishing markings are those in which **at least one** immediate transition is enabled. Since vanishing markings are not resided in for any non-zero time and firings are acted instantaneously, the priority of immediate transitions is always higher than that of timed transitions.

Since computers have limited resource, only bounded GSPNs, whose underlying reachability sets are finite, are considered. Under the condition that only a positive number of transitions can fire in a finite time with non-zero probability, there is exactly one Continuous Time Markov Chain (CTMC) that corresponds to a given GSPN [10].

2.1.2 Stochastic Reward Nets (SRNs)

Stochastic Reward Nets (SRNs) are based on GSPN but extend them further [3]. Some of the most prominent extensions are revisited in the following: priorities, guards, marking dependent arc multiplicity, marking-dependent firing rates, and reward rates defined at the net level.

Priorities: As mentioned in the previous section, priority is important when more than one transition is enabled at the same time. Although inhibitor arcs can be used to achieve priority relationships, for the purpose of simplifying the model description, explicit priorities can be assigned to transitions. Priorities are specified by assigning integer numbers to transitions. A transition is enabled only if there is no other transition with a higher priority enabled.

Guards: The guard functions are similar to the inhibitor arcs, but can use the entire state of the net rather than just the number of tokens in places. They determine when transitions are to be enabled. This feature provides a powerful means to simplify the graphical representation and to make SRNs easier to understand in a more general way compared to the use of inhibitor arcs.

Marking-Dependent Arc Multiplicity: This feature provides a way to change the structure of SRNs. For example, when a critical component of the system is down, the system is down. The way for us to represent the situation is to flush all places which have number of tokens representing available resources in the system. The example showing the use of this feature is in the section 2.4.5.

Marking-Dependent Firing Rates: The firing rate of a transition may depend in a rather general way based on the current marking of the net. In the implementation, there are two ways: one way is to use rate functions, which are similar to guard functions and reward rate functions; another way is to use the number of tokens in a chosen place multiplying the basic rate of the transition, which is called place-dependent firing rate. For the first situation, there is a SRN example of Markov Modulated Poisson Processes (MMPPs) [4] in the right part of Figure 2.5. The firing rate of the transition $T3$ depends on whether there is a token in the place $P1$. When there is one and only one token in $P1$, the firing rate is a_1; Otherwise, it is a_0. Since there is an inhibitor arc from $P1$ to $T1$, $P1$ can only have one token at most.

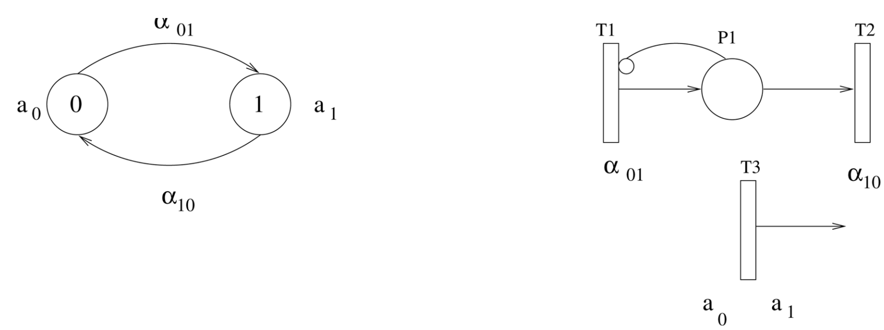


Figure 2.5: Extension of GSPN 2

The left part of Figure 2.5 shows the corresponding CTMC which decides the firing rate of the transition $T3$.

Reward Rate Specification: The basic output measures obtained from a SRN are the throughput of a transition and the mean number of tokens in a place. But that's far from enough. Normally, more general information, such as the probability that a place is empty while another one is full, or the sum of the number of tokens in a set of places, is necessary. Since it is at the net level rather than at the place level, reward rate functions are introduced.

Compared to GSPN, SRN provides more power and eases the work of translating real-world systems into analytic models. That's why SRN has been implemented in SHARPE.

2.2 How to solve

First, consider a computing system model (example 2.4.1) shown in Figure 2.8.

Next step, the SRN in Figure 2.8 is converted into the corresponding reachability graph. Figure 2.6 shows the reachability graph. Notice that vanishing markings are shown as dotted rectangle.

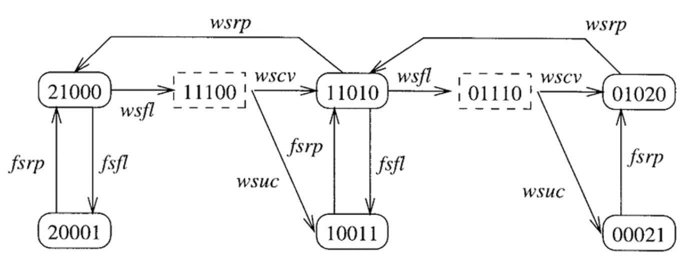


Figure 2.6: The Reachability Graph for the system in Figure 2.8

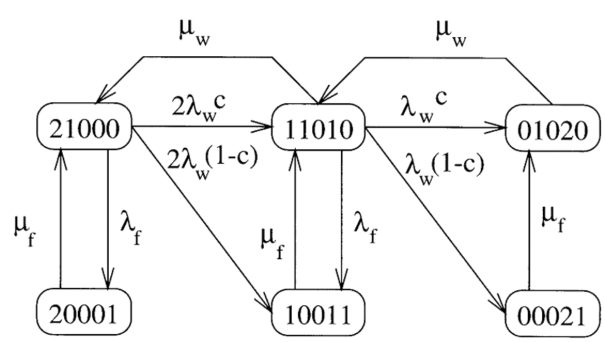


Figure 2.7: CTMC after deleting vanishing markings from Figure 2.6

Assign rates and probabilities to each arc in the reachability graph, and eliminate all vanishing markings. The corresponding Continuous Time Markov Chain is shown in Figure 2.7, where, respectively, λ_w and λ_f are the failure rates of each workstation and the file server, and μ_w and μ_f represent the repair rates of each workstation and the file server.

For transient analysis, *randomization* [15], sometimes called *uniformization*, is used to solve the problem. For steady-state analysis, Gauss-Seidel and Successive Over-Relaxation are used.

2.3 Implementation

2.3.1 Syntax for SRNs

The syntax for SRN models in SHARPE is as the following:

srn *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* {**guard** *expression* } {**priority** expression}>

<*transition_name* **placedep** *place_name expression* {**guard** *expression* } {**priority** expression}>

<*transition_name* **gendep** *expression* {**guard** *expression* } {**priority** expression} >

}

end

∗ section 3: immediate transition names, types and weights

{

<*transition_name* **ind** *expression*{**guard** *expression* } {**priority** expression}>

<*transition_name* **placedep** *place_name expression*{**guard** *expression* } {**priority** expression}>

<*transition_name* **gendep** *expression* {**guard** *expression* } {**priority** 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

where, *param_list* is:

name, name, ..., name

name, *trans_name* and *place_name* are all symbols; *expression* is a mathematical expression that could contain function calls; *ind* means that the transition's firing rate is not dependent on the current marking of the net; *placedep* means that the transition's firing rate depends on the number of tokens in the specific place mentioned and the expression assigned to it; and *gendep* means that the firing rate depends on the marking-dependent function referenced in the corresponding *expression*.

2.3.2 New built-in functions

Marking-dependent and rate-dependent functions

The following functions are used only within reward functions, guard functions, rate functions, and arc cardinality functions for SRN models.

- $\#(place\_name)$

  Returns the number of tokens in a place with the given *place_name*.

- $?(trans\_name)$

  Returns the boolean (true or *false*) value depending on whether the given transition *trans_name* is enabled.

- $Rate(trans\_name)$

  Returns the rate of the given transition *trans_name*; if disabled, return 0.

System analysis functions

In addition to the system analysis functions used for GSPN, three new system analysis functions have been introduced to deal with the power of SRN models.

- **srn_exrss** *(sys_name ; reward_func_name{; arglist})*

  Calculates the steady-state expected value of the reward function *reward_func_name*.

- **srn_exrt** *(t, sys_name; reward_func_name{;arglist})*

  Calculates the expected value of the reward function *reward_func_name* at time t.

- **srn_cexrt** *(t, sys_name; reward_func_name{; arglist})*

  Calculates the cumulative expected value of the reward function *reward_func_name* over the interval $(0, t]$.

- **srn_ave_cexrt** *(t, sys_name; reward_func_name{; arglist})*

  Calculates the average cumulative expected value of the reward function *reward_func_name* over the interval $(0, t]$.

- **mtta** *(sys_name {; arglist})*

  Calculates the mean time to absorption for the SRN named *sys_name*. The function should be used only when the underlying CTMC has absorbing states. (See example C.4.1)

- **srn_cexrinf** *(sys_name; reward_func_name{; arglist})*

  Calculates the cumulative expected value of the reward function *reward_func_name* until absorption for the corresponding CTMC of the SRN system *sys_name*. The CTMC must have absorbing states. (See example C.4.1)

where, $arglist$ is

expression, expression, ..., expression

Mathematical functions

All the following functions can be used within *expression*s, for all models including the SRN model.

- **acos** (*expression*)

Calculates the arccosine.

- **asin** (*expression*)

  Calculates the arcsine.

- **atan** (*expression*)

  Calculates the arctangent.

- **ceil** (*expression*)

  Calculates the ceiling of a value.

- **cos** (*expression*)

  Calculates the cosine.

- **fabs** (*expression*)

  Calculates the absolute value.

- **floor** (*expression*)

  Calculates the floor of a value.

- **ln** (*expression*)

  Calculates natural logarithm.

- **max** (*expression*, *expression*)

  Compares two values and returns the larger one.

- **min** (*expression*, *expression*)

  Compares two values and returns the smaller one.

- **sin** (*expression*)

  Calculates sine.

- **sqrt** (*expression*)

  Finds square root.

- **tan** (*expression*)

  Calculates the tangent.

- **weibull** (*expression1*, *expression2*, *expression3*)

  Calculates the Weibull distribution function $1 - e^{-\frac{expression1^{expression3}}{expression2}}$

2.3.3 Syntax extensions

User defined function

Now, SHARPE supports either the old way of defining a function:

func (*param_list*) *expression*

or the new way:

func (*param_list*)

<statement>

end

If-statement has been added:

if *bool_expression*

<statement>

{ < **elseif** *bool_expression*

<statement> >}

{**else**

<statement>}

end

where *statement* can be

expression | **bind** *var_name expression* | **epsilon** *epsilon_type expression* | **if**_*statement*

Detailed examples are provided in section 2.4.

Fixed point iteration

Suppose we have one SRN model M_1. The firing rate R_1 of a transition T_1 is the same as the throughput of another transition T_2 [5]. Since we don't know the firing rate of T_1, fixed point iteration has to be used:

1. Set error bound e as a small real number, normally $1e-7$ in SHARPE.
2. Initialize the firing rate R_1^0 of T_1 to a reasonable value.
3. Set $k = 1$
4. Execute M_1, compute the throughput $T2_{throughput}$ of T_2.
5. Set $R_1^k = T2_{throughput}$.
6. If $|R_1^k - R_1^{k-1}|/R_1^{k-1} < e$, then stop, else set $k = k+1$ and goto step 4.

Under a very general condition, the solution always exists, but the uniqueness of the solution is not guaranteed [2]. However, in many of practical problems, result is often unique, so the justification is enough for the practical use of fixed point iterations.

To support fixed point iteration, **while**-statement has been introduced:

while *bool_expression*

<statement>

end

where *statement* can be

expr *expression*{,*expression* ...} | **bind** *var_name expression* | **epsilon** *epsilon_type expression* | **if**_*statement* | *loop* | **while**_*statement*

There is an example of fix-point iteration in section 2.4.9. Also, an example of **while**-statement has been included in section 2.4.10.

2.4 SRN Examples

2.4.1 Two workstations, one file server system

Description

A system contains 2 workstations and 1 file server (Figure 2.8) . Suppose the network is fault-free, and the whole system is working as long as there is one workstation and the file server is operational. So, the initial number of tokens in the place *wsup* is 2 and in the place *fsup* is 1. The file server has higher repair priority than the two workstations(see the inhibitor arc from the place *fsdn* to the transition *wsrp* in Figure 2.8). Also, when the whole system is down, currently operational workstations or file server don't go down any more(see the inhibitor arcs from *fsdn* and *wsdn* to the transitions *wsfl* and *fsrp* in Figure 2.8). We also have the assumption that, when a workstation fails, with probability c, the failure is not detected, leading to the corruption and the failure of the file-server. That's why we have immediate transitions *wscv* and *wsuc*.

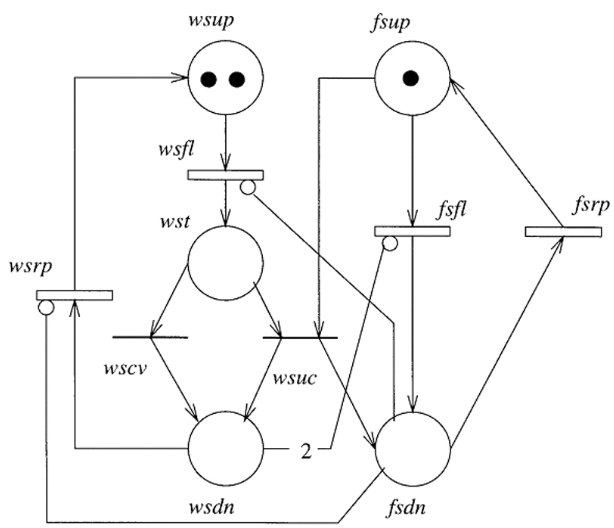


Figure 2.8: Two workstation, one file server system with non-perfect failure detect

Features

- Reward function to compute expected values.
- Transient analysis

SHARPE File — *srn/wfs.txt*

```
format 8

func avail()
if ((#(wsup) > 0) and (#(fsup) == 1))
1
```

```
else
0
end
end


srn wfs (c)
* Places
wsup    2
fsup    1
wst    0
wsdn    0
fsdn    0
end
* Timed transitions
wsfl    placedep    wsup    0.0001
fsfl    ind    0.00005
wsrp    ind    1.0
fsrp    ind    0.5
end
* Immediate transitions
wscv    ind    c
wsuc    ind    1 −c
end
* Input arcs
wsup    wsfl    1
fsup    fsfl    1
fsup    wsuc    1
wst    wscv    1
wst    wsuc    1
wsdn    wsrp    1
fsdn    fsrp    1
```

```
end
* Output arcs
wsfl    wst    1
wsrp    wsup    1
fsfl    fsdn    1
fsrp    fsup    1
wscv    wsdn    1
wsuc    wsdn    1
wsuc    fsdn    1
end
* Inhibitor arcs
fsdn    wsfl    1
fsdn    wsrp    1
wsdn    fsfl    2
end

* Obtain results
loop c, 0.7, 0.9, 0.1
   loop t, 1, 10, 1
      expr srn_exrt(t, wfs; avail; c)
   end
   expr srn_exrt(20, wfs; avail;c)
end


end
```

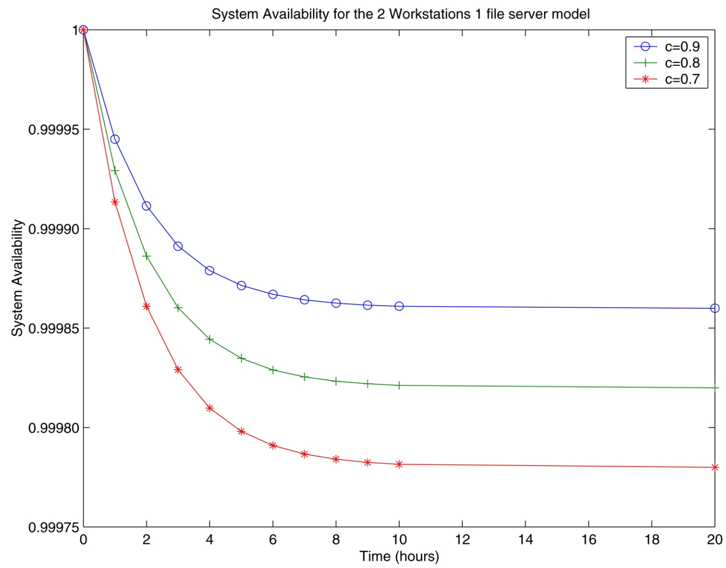


Figure 2.9: Graph result for example 2.4.1

The result is shown graphically in Figure 2.9

2.4.2 Molloy's example

Source

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

Description

The net is shown in Figure 2.10

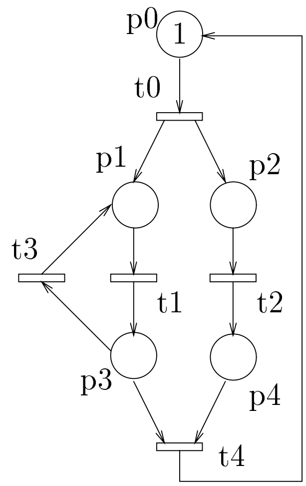


Figure 2.10: SRN for Example 2.4.2

Features

- Reward based functions to compute expected values.
- Default measures
- Steady-state analysis

SHARPE File — *srn/ex1.txt*

```
echo M. K. Molloy, Performance Analysis Using Stochastic Petri Nets,
echo IEEE Trans. Comput., C−31(9), Sept. 1982, 931−917

format 8


srn example1()
p0   1
p1   0
```

```
p2  0
p3  0
p4  0
end
t0  ind  1.0
t1  ind  3.0
t2  ind  7.0
t3  ind  9.0
t4  ind  5.0
end
end
p0  t0  1
p1  t1  1
p2  t2  1
p3  t3  1
p3  t4  1
p4  t4  1
end
t0  p1  1
t0  p2  1
t1  p3  1
t2  p4  1
t3  p1  1
t4  p0  1
end
end

* REWARD functions

func  ef0()  #(p0)

func  ef1()  #(p1)
```

func ef2() Rate(t2)

func ef3() Rate(t3)

func eff() Rate(t1)*1.8+#(p3)*0.7

* Obtain results

expr srn_exrss(example1; ef0), srn_exrss(example1; ef1), srn_exrss(example1; ef2), srn_exrss(example1; ef3), srn_exrss(example1; eff)

end

2.4.3 Software Performance Analysis

Description

This example models the following piece of software:

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

The corresponding SRN model is shown in Figure 2.11.

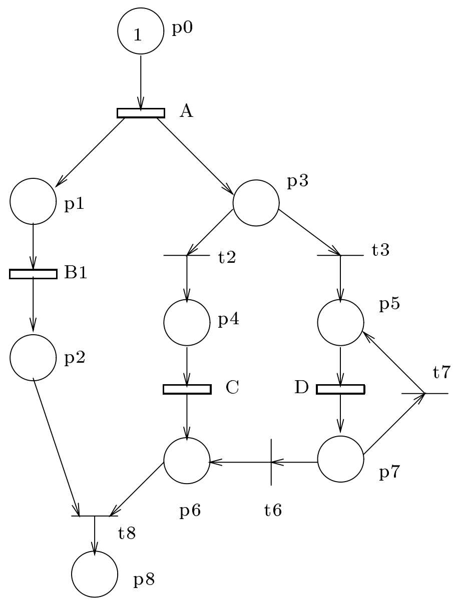


Figure 2.11: SRN for Example 2.4.3

Features

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

SHARPE File — *srn/ex2.txt*

```
echo Software Performance Analysis
echo A: Statements;
echo PARBEGIN
echo     B1: statements;
echo     B2: IF (cond1) THEN
echo            C: statements;
```

```
echo        ELSE
echo            DO
echo           D: statements;
echo           WHILE (cond2);
echo        END IF
echo PAREND

format 8

bind
rate0    1.0
rate1    0.3
prob2    0.4
prob3    0.6
rate4    0.2
rate5    7.0
prob6    0.05
prob7    0.95
prob8    1.0
end

srn ex2()
∗ Places
P0    1
P1    0
P2    0
P3    0
P4    0
P5    0
P6    0
P7    0
P8    0
```

end

∗ Timed transitions

A ind rate0

B1 ind rate1

C ind rate4

D ind rate5

end

∗ Immediate transitions

t2 ind prob2

t3 ind prob3

t6 ind prob6

t7 ind prob7

t8 ind prob8

end

∗ Input arcs

P0 A 1

P1 B1 1

P3 t2 1

P3 t3 1

P4 C 1

P5 D 1

P7 t6 1

P7 t7 1

P2 t8 1

P6 t8 1

end

∗ Output arcs

A P1 1

B1 P2 1

t2 P4 1

t3 P5 1

C P6 1

```
D    P7    1
t6   P6    1
t7   P5    1
A    P3    1
t8   P8    1
end
* Inhibitor arcs
end

func   rfunc()   #(P8)

echo probability of completion
loop   i, 1, 10
   srn_exrt(i, ex2; rfunc)
end
loop   i, 10, 20, 2
   srn_exrt(i, ex2; rfunc)
end
loop   i, 20, 50, 5
   srn_exrt(i, ex2; rfunc)
end

end
```

2.4.4 $M/M/m/b$ queue

Description

This example models a finite-buffer $M/M/m/b$ queue shown in Figure 2.12. The corresponding SRN is shown in Figure 2.13.

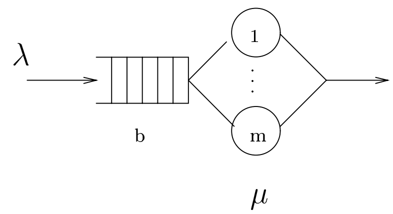
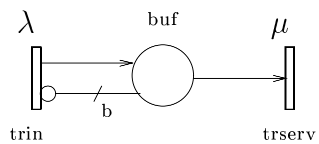


Figure 2.12: The $M/M/m/b$ Queue.



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

Figure 2.13: SRN for Example 2.4.4

Features

- Both steady-state and transient analysis.
- Marking dependent firing rates.
- Reward functions.

SHARPE File — *srn/ex3.txt*

echo M/M/m/b queue model

format 8

bind

lambda 0.90

mu 0.10

∗number of buffers

```
b  2
*number of servers
m  2
end

* RATE function
func  rate_serv()
if (#(buf) < m)
#(buf)*mu
else
m*mu
end
end

srn example3()
* Places
buf  0
end
* Timed transitions
trin  ind  lambda
trserv  gendep  rate_serv()
end
* Immediate transitions
end
* Input arcs
buf  trserv  1
end
* Output arcs
trin  buf  1
end
* Inhibitor arcs
buf  trin  b
```

```
end

* REWARD functions
func    qlength1()    #(buf)

func    util1()    ?(trserv)

func    tput1()    Rate(trserv)

func    probrej()
if (#(buf) == b)
1
else
0
end
end

func    probempty()
if (#(buf)==0)
1
else
0
end
end

func    probhalffull()
if (#(buf) == b/2)
1
else
0
end
end
```

```
* Obtain results
expr srn_exrss(example3; qlength1), srn_exrss(example3; tput1), srn_exrss(example3; util1), srn_exrss(example3; probrej), srn_exrss(example3; probempty), srn_exrss(example3; probhalffull)

loop t, 0.1, 1.0, 0.1
expr srn_exrt(t, example3; qlength1), srn_exrt(t, example3; tput1), srn_exrt(t, example3; util1), srn_exrt(t, example3; probrej), srn_exrt(t, example3; probempty), srn_exrt(t, example3; probhalffull)
end

loop t, 1.0, 10.0, 1.0
expr srn_exrt(t, example3; qlength1), srn_exrt(t, example3; tput1), srn_exrt(t, example3; util1), srn_exrt(t, example3; probrej), srn_exrt(t, example3; probempty), srn_exrt(t, example3; probhalffull)
end

end
```

2.4.5 C.mmp system performability analysis

Source

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

Description

This example models the C.mmp system designed at CMU. The architecture of the system is shown in Figure 2.14. The corresponding SRN model is shown in Figure 2.15.

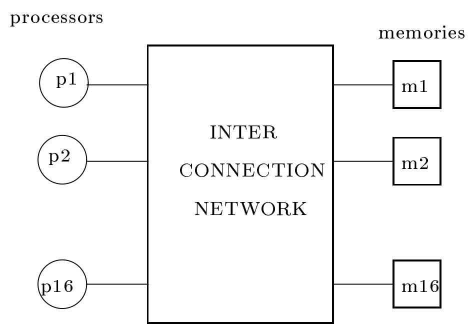


Figure 2.14: The C.mmp Architecture.

Features

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

SHARPE File — *srn/ex4.txt*

echo C.mmp system performability analysis

echo J.T. Blake, A.L. Reibman and K.S. Trivedi,

echo Sensitivity Analysis of Reliability and Performability

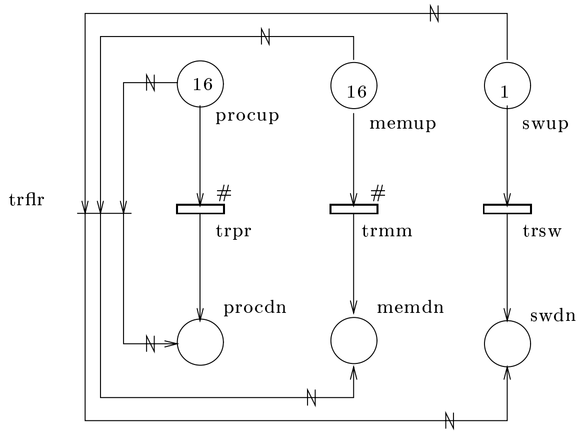
echo Measures for Multiprocessor Systems,

echo Proc. 1988 ACM SIGMETRICS, Santa Fe, NM, 1988

format 8

∗ Munimum number of proc/mem needed 1≤k≤16

bind k 2



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

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

Figure 2.15: SRN for Example 2.4.5.

∗ GUARD function

func entrflr()

if (#(procup) == 0 and #(memup) == 0 and #(swup) == 0)

0

elseif (#(procup) < k or #(memup) < k or #(swup) == 0)

1

else

0

end

end

∗ ARC CARDNALITY functions

func apfl() #(procup)

```
func    amfl()    #(memup)

func    asfl()    #(swup)

srn example4()
∗ Places
procup    16
procdn    0
memup    16
memdn    0
swup    1
swdn    0
end
∗ Timed transitions
trpr    placedep    procup    0.0000689
trmm    placedep    memup    0.000224
trsw    ind    0.0002202
end
∗ Immediate transitions
trflr    ind    1.0    guard    entrflr()    priority    100
end
∗ Input transitions
procup    trpr    1
memup    trmm    1
swup    trsw    1
procup    trflr    apfl()
memup    trflr    amfl()
swup    trflr    asfl()
end
∗ Output transitions
trpr    procdn    1
trmm    memdn    1
```

```
trsw    swdn    1
trflr    procdn    apfl()
trflr    memdn    amfl()
trflr    swdn    asfl()
end
∗ Inhibitor arcs
end

∗ REWARD functions
func    reliab()
if (#(procup) ≥ k and #(memup) ≥ k and #(swup) == 1)
1
else
0
end
end

func    reward_rate()
if (#(procup) ≥ k and #(memup) ≥ k and #(swup) == 1)
if (#(procup) > #(memup))
bind l    #(memup)
bind m    #(procup)
else
bind m    #(memup)
bind l    #(procup)
end
bind temp    (1.0−(1.0/m))ˆl
m∗(1.0 − temp)
else
0
end
end
```

∗ Obtain results

```
loop t, 500.0, 5000.0, 500.0
expr    srn_exrt(t, example4; reliab), srn_exrt(t, example4; reward_rate), srn_cexrt(t, example4; reward_rate)
end

end
```

Result (Figure 2.16)

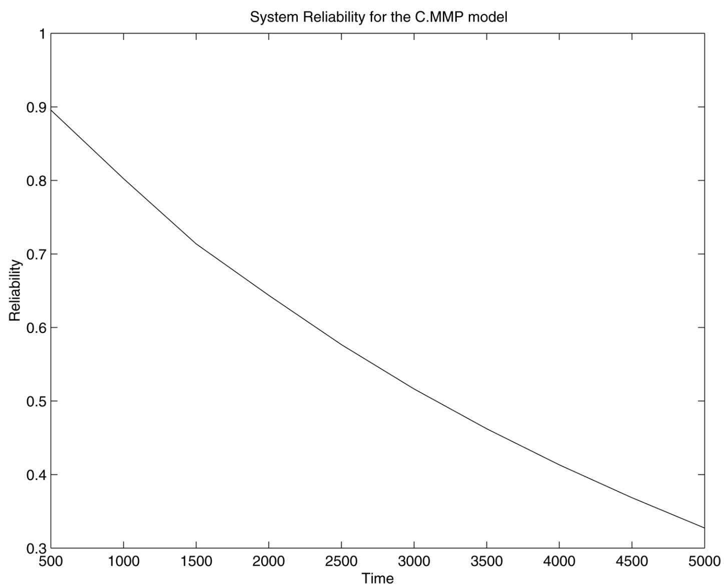


Figure 2.16: Graph result for example 2.4.5

2.4.6 Database system availability analysis

Source

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

Description

This example is a model of a database system shown in Figure 2.17.

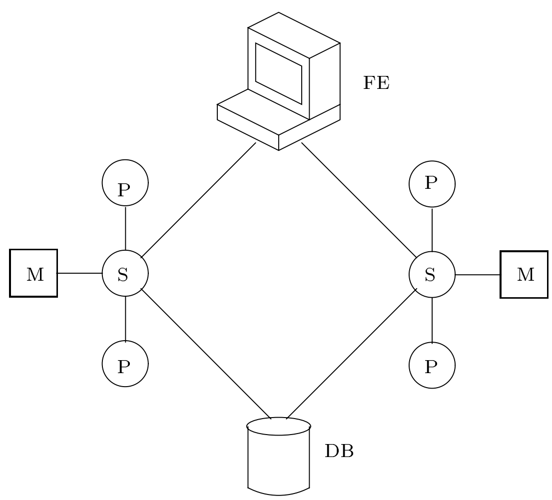
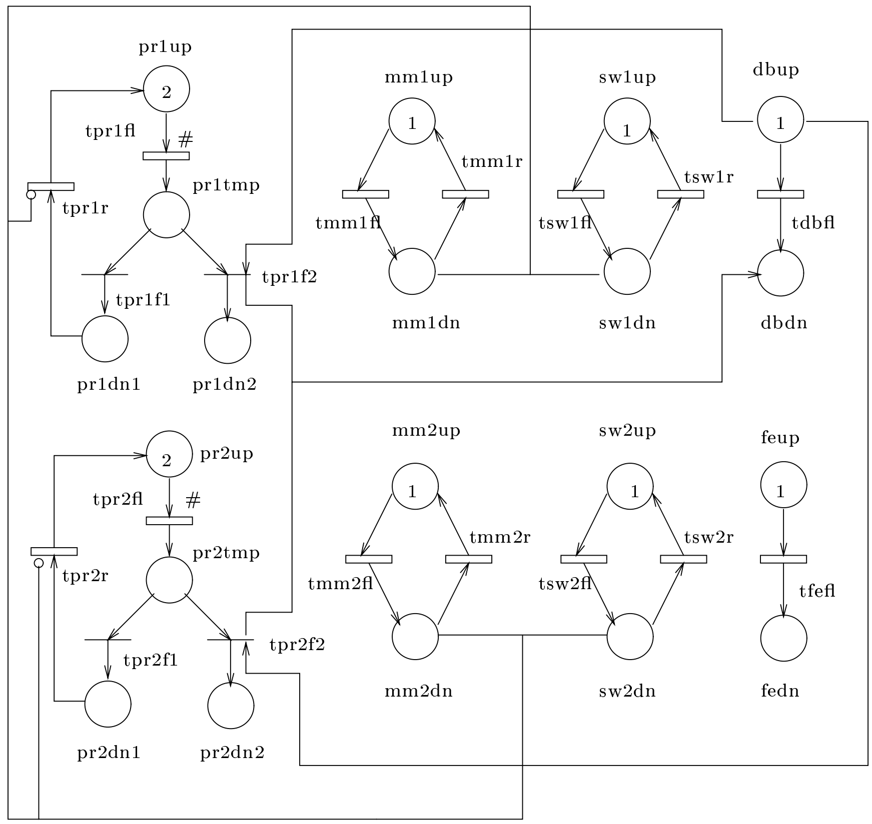


Figure 2.17: The Database System Architecture.

The system consists of a front end (FE), a database (DB) and two processing sub-systems. Each processing sub-system consists of two processors (P), a memory (M) and a switch (S). For the system to be functional, we need at least one of the processing sub-systems to be operational. The database and the front-end should also be operational. The



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

Figure 2.18: SRN for Example 2.4.6.

processing sub-system is functional as long as the memory, the switch and at least one of the processors is functional. When a processor fails, with probability c it fails without disturbing the system. However, with probability $1 - c$ the failing processor corrupts the database causing it to fail and consequently rendering the system un-operational. The processors, memories and switches can be repaired while the system is up. The memories and switches receive priority over the processors for repair. The corresponding SRN model is shown in Figure 2.18.

Features

- Guard function.
- Reward based functions.
- Transient analysis.

SHARPE File — *srn/ex5.txt*

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

format 8

epsilon basic 1.0e−10

bind    coverage    0.99
bind    count       0

* GUARD functions
func enall()
if (#(dbup)==0)
0
elseif (#(feup)==0)
0
elseif (((#(mm1up)==0) or (#(sw1up)==0) or (#(pr1up)==0)) and ((#(mm2up)==0) or (#(sw2up)==0) or (#(pr2up)==0)))
0
else
```

```
1
end
end

srn example5()
∗ Places
∗ First processing subsystem
mm1up    1
sw1up    1
pr1up    2
mm1dn    0
sw1dn    0
pr1tmp    0
pr1dn1    0
pr1dn2    0
∗ Second processing subsystem
mm2up    1
sw2up    1
pr2up    2
mm2dn    0
sw2dn    0
pr2tmp    0
pr2dn1    0
pr2dn2    0
∗ Database
dbup    1
dbdn    0
∗ Frontend
feup    1
fedn    0
end
∗ Timed transitions
```

tmm1fl ind 1000./2400. guard enall()

tsw1fl ind 1000./2400. guard enall()

tpr1fl placedep pr1up 1000./2400. guard enall()

tmm1r ind 1000. guard enall()

tsw1r ind 1000. guard enall()

tpr1r ind 1000. guard enall()

tmm2fl ind 1000./2400. guard enall()

tsw2fl ind 1000./2400. guard enall()

tpr2fl placedep pr2up 1000./2400. guard enall()

tmm2r ind 1000. guard enall()

tsw2r ind 1000. guard enall()

tpr2r ind 1000. guard enall()

tdbfl ind 1000./2400. guard enall()

tfefl ind 1000./2400. guard enall()

end

∗ Immediate transitions

tpr1f1 ind coverage priority 100

tpr1f2 ind 1.0−coverage priority 100

tpr2f1 ind coverage priority 100

tpr2f2 ind 1.0−coverage priority 100

end

∗ Input arcs

mm1up tmm1fl 1

sw1up tsw1fl 1

pr1up tpr1fl 1

pr1tmp tpr1f1 1

pr1tmp tpr1f2 1

dbup tpr1f2 1

mm1dn tmm1r 1

sw1dn tsw1r 1

pr1dn1 tpr1r 1

mm2up tmm2fl 1

sw2up tsw2fl 1

pr2up tpr2fl 1

pr2tmp tpr2f1 1

pr2tmp tpr2f2 1

dbup tpr2f2 1

mm2dn tmm2r 1

sw2dn tsw2r 1

pr2dn1 tpr2r 1

dbup tdbfl 1

feup tfefl 1

end

∗ Output arcs

tmm1fl mm1dn 1

tsw1fl sw1dn 1

tpr1fl pr1tmp 1

tpr1f1 pr1dn1 1

tpr1f2 pr1dn2 1

tpr1f2 dbdn 1

tmm1r mm1up 1

tsw1r sw1up 1

tpr1r pr1up 1

tmm2fl mm2dn 1

tsw2fl sw2dn 1

tpr2fl pr2tmp 1

tpr2f1 pr2dn1 1

tpr2f2 pr2dn2 1

tpr2f2 dbdn 1

tmm2r mm2up 1

tsw2r sw2up 1

tpr2r pr2up 1

tdbfl dbdn 1

tfefl fedn 1

```
end
* Inhibitor arcs
mm1dn    tpr1r    1
mm2dn    tpr1r    1
sw1dn    tpr1r    1
sw2dn    tpr1r    1
mm1dn    tpr2r    1
mm2dn    tpr2r    1
sw1dn    tpr2r    1
sw2dn    tpr2r    1
end

* REWARD function
func    reliab()
if (#(dbup)==0)
0.0
elseif (#(feup)==0)
0.0
elseif (((#(mm1up)==0) or (#(sw1up)==0) or (#(pr1up)==0)) and ((#(mm2up)==0) or (#(sw2up)==0) or (#(pr2up)==0)))
0.0
else
1.0
end
end

* Obtain results
loop t, 0.01, 0.1, 0.01
   expr srn_exrt(t, example5; reliab)
end
*echo error cumulated
loop t, 0.1, 1, 0.1
   expr srn_exrt(t, example5; reliab)
```

```
end

end
```

2.4.7 ATM network under overload

Source

Chang-Yu Wang, D. Logothetis, K.S. Trivedi and I. Viniotis, Transient Behavior of ATM Networks under Overloads, *Proceedings of the IEEE INFOCOM 96*, San Francisco, CA, pp. 978-985, March 1996.

Description

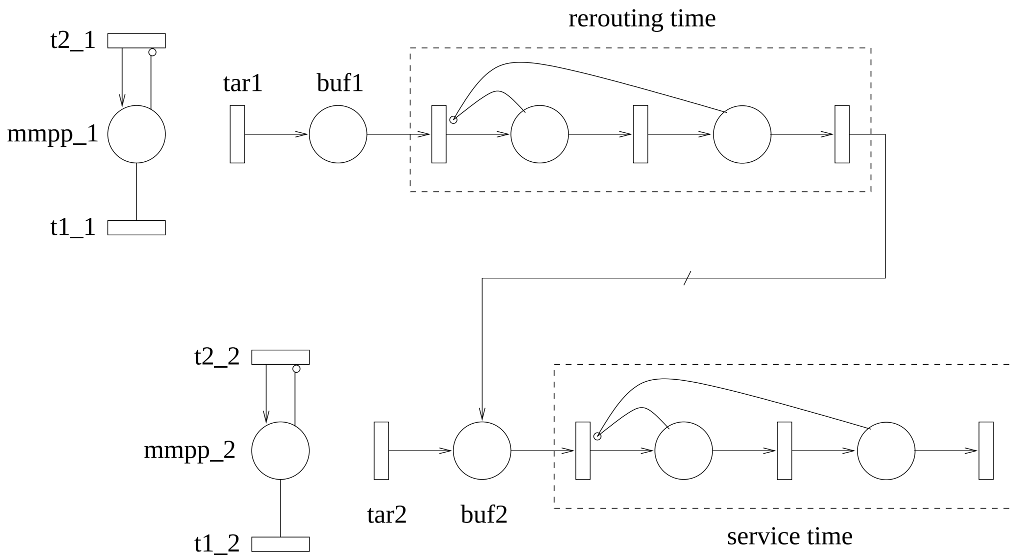
This example models ATM (Asynchronous Transfer Mode) networks under overloads. The SRN is shown in Figure 2.19.

Features

- Transient analysis.
- Marking dependent firing rates.
- Guard functions.
- Reward functions.

SHARPE File — *srn/ex6.txt*

```
echo ATM network under overload
echo Chang–Yu Wang, D. Logothetis, K.S. Trivedi and I. Viniotis,
```



Transition	Rate Function	Guard Function
tar_1	if $(\#mmpp_1)$ λ_1^1 else λ_2^1	$(\#buf_1 + \sum_i \#P_{reroute[i]}) < K_1$
tar_2	if $(\#mmpp_2)$ λ_1^2 else λ_2^2	$(\#buf_2 + \sum_i \#P_{serv}[i]) < K_2$

Figure 2.19: SRN for Example 2.4.7

```
echo Transient Behavior of ATM Networks under Overloads,
echo Proceedings of the IEEE INFOCOM 96, San Francisco, CA,
echo pp. 978−985, March 1996.

format 8

bind
a1    0.0269163
a2    0.0269163
b1    0.00672908
b2    0.00672908
lambda11    1.5058
lambda21    1.5058
lambda12    0.00301161
lambda22    0.00301161
r1    5
```

```
r2    5
mu1    2.73
mu2    2.73
K1    16
K2    16
e    0.0001
end

*REWARD Functions
func    Qlen1() #(buf1)+(#(Er_token1)+#(Er_stage1))/r1

func    Earrival()
if (#(mmpp_2)<> 0)
bind ret_val lambda21
else
bind ret_val lambda22
end
if (#(Er_token1)==1)
bind ret_val ret_val+r1/mu1
end
ret_val
end

func    Qlen2() #(buf2)+(#(Er_token2)+#(Er_stage2))/r1

func    ELR()
if ((Qlen2()+e)≥K2)
if (#(mmpp_2)<>0)
bind ret_val lambda21
else
bind ret_val lambda22
end
```

```
if (#(Er_token1)==1)
bind ret_val ret_val+r1/mu1
end
ret_val
else
0
end
end

func    PFull()
if (Qlen2()+e)≥K2
1.0
else
0
end
end

∗ GUARD Functions
func    gar2()
if (Qlen2()+e)<K2
1
else
0
end
end

func    gar1()
if ((Qlen1()+e)<K1)
1
else
0
end
```

```
end

* RATE Functions
func    REr1() r1/mu1

func    Rar1()
if (#(mmpp_1)>0)
lambda11
else
lambda12
end
end

func    REr2() r2/mu2

func    Rar2()
if (#(mmpp_2)>0)
lambda21
else
lambda22
end
end

* CARDINALITY Functions
func    R2() r2

func    dep12()
if ((K2-Qlen2()+e)<1)
0
else
1
end
end
```

```
func    R1() r1

srn example6()
* Places
mmpp_1    1
mmpp_2    1
buf1    0
Er_token1    0
Er_stage1    0
buf2    0
Er_token2    0
Er_stage2    0
end
* Timed Transitions
t2_1    ind    b1
t2_2    ind    b2
t1_1    ind    a1
t1_2    ind    a2
tar1    gendep    Rar1()    guard    gar1()
Er_trans1    ind    REr1()
tar2    gendep    Rar2()    guard    gar2()
Er_trans2    ind    REr2()
end
* Immediate Transitions
Er_in1    ind    1.    priority    20
Er_out1    ind    1.    priority    20
Er_in2    ind    1.    priority    20
Er_out2    ind    1.    priority    20
end
* Input arcs
mmpp_1    t1_1    1
```

```
mmpp_2    t1_2    1
buf1    Er_in1    1
Er_token1    Er_trans1    1
Er_stage1    Er_out1        R1()
buf2    Er_in2    1
Er_token2    Er_trans2    1
Er_stage2    Er_out2        R2()
end
* Output arcs
t2_1    mmpp_1    1
t2_2    mmpp_2    1
tar1    buf1    1
Er_in1        Er_token1    R1()
Er_trans1    Er_stage1    1
Er_out1    buf2    dep12()
tar2    buf2    1
Er_in2        Er_token2    R2()
Er_trans2    Er_stage2    1
end
* Inhibitor arcs
mmpp_1    t2_1    1
mmpp_2    t2_2    1
Er_token1    Er_in1    1
Er_stage1    Er_in1    1
Er_token2    Er_in2    1
Er_stage2    Er_in2    1
end

* Obtain results
loop t, 10.0, 200.0, 10.0
    expr srn_exrt(t, example6; Qlen1)
    expr srn_exrt(t, example6; Qlen2)
```

```
    expr srn_exrt(t, example6; ELR)
    expr srn_exrt(t, example6; PFull)
    expr srn_exrt(t, example6; Earrival)
end

end
```

Result (Figure 2.20)

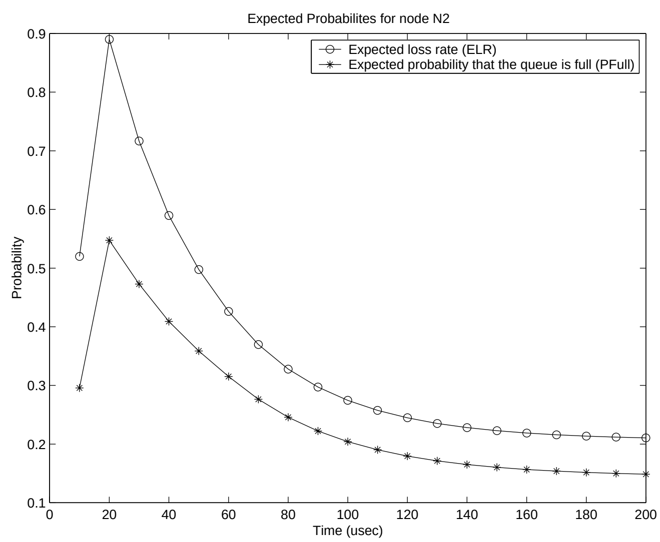


Figure 2.20: Partial graph result for example 2.4.7

2.4.8 Criticality Importance and Birnbaum Importance

Source

R. M. Fricks and K. S. Trivedi, On Computing Importance Measures Using Reward Models, *VII Simposio de Computadores Tolerantes a Falhas (VII SCTF)*, pp. 169 – 183, Campina Grande, Brazil, Jul. 1997.

Description

A novel technique for computing importance measures in state space dependability models is introduced here. Specifically, reward functions in a Markov reward model are utilized for this purpose, in contrast to the common method of computing importance measures through combinatorial models and structure functions. The following simple example is used to show how to calculate Criticality Importance and Birnbaum Importance.

Features

- Reward based measures.

SHARPE File — *srn/ex7.txt*

```
echo Criticality Importance and Birnbaum Importance
echo R.M. Fricks and K. S. Trivedi,
echo On Computing Importance Measures Using Reward Models,
echo VII Simposio de Computadores Tolerantes a Falhas (VII SCTF),
echo pp. 169−183, Campina Grande, Brazil, Jul. 1997.

format 8

* REWARD RATE FUNCTIONS

* Criticality
func    Q1()
if (#(p1) == 1)
1
else
0
```

```
end
end

func  Q2()
if (#(p2) == 1)
1
else
0
end
end

func  Q3()
if (#(p3) == 1)
1
else
0
end
end

func  Q()
if (Q1() + Q2() + Q3() ≥2)
1
else
0
end
end

∗ Birnbaum
func  g11()
if (1.0+Q2() + Q3() ≥ 2)
1
else
```

```
0
end
end

func    g10()
if (Q2() + Q3() ≥ 2)
1
else
0
end
end

func    g21()
if (Q1() +1.0 + Q3() ≥ 2)
1
else
0
end
end

func    g20()
if (Q1() + Q3() ≥ 2)
1
else
0
end
end

func    g31()
if (Q1()+Q2() + 1.0 ≥ 2)
1
else
```

```
0
end
end

func    g30()
if (Q1()+Q2()  ≥ 2)
1
else
0
end
end

srn    example7()
∗ Places
p1    0
p2    0
p3    0
end
∗ Timed transitions
t1    ind    0.001
t2    ind    0.002
t3    ind    0.003
end
∗ Immediate transitions
end
∗ Input arcs
end
∗ Output arcs
t1    p1    1
t2    p2    1
t3    p3    1
end
```

```
* Inhibitor arcs
p1   t1   1
p2   t2   1
p3   t3   1
end

* Obtain results
bind
t   20.
b1   srn_exrt(t, example7; g11) − srn_exrt(t, example7; g10)
b2   srn_exrt(t, example7; g21) − srn_exrt(t, example7; g20)
b3   srn_exrt(t, example7; g31) − srn_exrt(t, example7; g30)
q   srn_exrt(t, example7; Q)
end

expr   b1, b2, b3, b1*srn_exrt(t, example7; Q1)/q, b2*srn_exrt(t, example7; Q2)/q, b3*srn_exrt(t, example7; Q3)/q

end
```

2.4.9 Channel recovery scheme in a cellular network

Source

Y. Ma, C. W. Ro and K. S. Trivedi, Performability Analysis of Channel Allocation with Channel Recovery Strategy in Cellular Network, *Proceedings of IEEE 1998 International Conference on Universal Personal Communications (ICUPC'98)*, Florence, Italy, 5-9 October, 1998.

Description

The net is shown in Figure 2.21

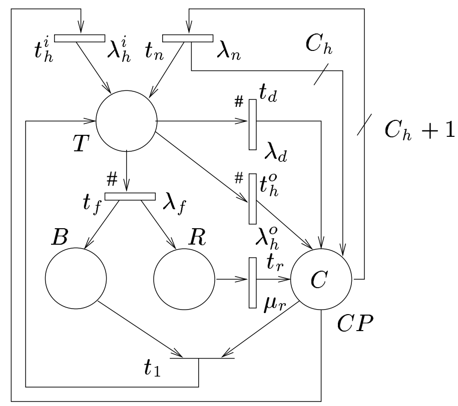


Figure 2.21: SRN for a channel recovery scheme in a cellular network.

Features

- Fixed point iteration. The handoff arrival rate (λ_h^i) of transition t_h^i equals to the throughput of transition t_h^o, which is used to represent the departure of handoff calls.
- Reward based functions to compute expected values.
- Default measures
- Transient analysis

SHARPE File — *srn/ex8.txt*

echo Y. Ma, C. W. Ro and K. S. Trivedi, Performability Analysis of Channel

echo Allocation with Channel Recovery Strategy in Cellular Network,

echo Proceedings of IEEE 1998 International Conference on Universal Personal echo Communications (ICUPC 1998), Florene, Italy, 5−9 October, 1998.

```
format 8

bind
MAX_ITERATIONS    6
MAX_ERROR     1e−7
t_channel    28
g_c        1
* New call arrival rate
lam_n        10
* handoff every 5 minutes
lam_h_o        0.33
* Handoff_in rate
lam_h_i        0.2
* call duration: 120 seconds
lam_d        0.5
lam_f        0.000016677
mu_r        0.0167
end

srn icupc98 ()
* Places
T    0
B    0
R    0
CP    t_channel
end
* Timed transitions
t_n    ind    lam_n
t_h_i    ind    lam_h_i
t_d    placedep    T    lam_d
```

t_f placedep T lam_f

t_h_o placedep T lam_h_o

t_r ind mu_r

end

∗ Immediate transitions

t_1 ind 1.0 priority 100

end

∗ Input arcs

CP t_n g_c + 1

CP t_h_i 1

T t_h_o 1

T t_d 1

T t_f 1

R t_r 1

B t_1 1

CP t_1 1

end

∗ Output arcs

t_n T 1

t_n CP g_c

t_h_i T 1

t_h_o CP 1

t_d CP 1

t_f B 1

t_f R 1

t_r CP 1

t_1 T 1

end

∗ Inhibitor arcs

end

∗ REWARD rate functions

```
func    BH()
if (#(CP) == 0)
1.0
else
0.0
end
end

func    BN()
if (#(CP) ≤ g_c)
1.0
else
0.0
end
end

func    ACh()    #(CP)

func    hotput()    Rate(t_h_o)

func    ftput2()    Rate(t_f)

func    fnum()        #(B)

bind i        0
bind err    1

while (i < MAX_ITERATIONS and err > MAX_ERROR)
bind tp    srn_exrss(icupc98; hotput)
bind err fabs((lam_h_i − tp)/tp)
bind i        i + 1
if (i < MAX_ITERATIONS)
```

```
bind lam_h_i    tp
end
end

expr   srn_exrss(icupc98; BH)
expr   srn_exrss(icupc98; BN)
expr   srn_exrss(icupc98; ACh)
expr   srn_exrss(icupc98; fnum)/srn_exrss(icupc98; ftput2)

end
```

Result File — *srn/ex8.txt.out*

```
* Y. Ma, C. W. Ro and K. S. Trivedi, Performability Analysis of Channel
* Allocation with Channel Recovery Strategy in Cellular Network,
* Proceedings of IEEE 1998 International Conference on Universal Personal
* Communications (ICUPC 1998), Florene, Italy, 5-9 October, 1998.
   tp <- 4.054972
   err <- 0.950678
   i <- 1.000000
      lam_h_i <- 4.054972
   tp <- 5.557387
   err <- 0.270346
   i <- 2.000000
      lam_h_i <- 5.557387
   tp <- 6.098202
   err <- 0.088684
   i <- 3.000000
      lam_h_i <- 6.098202
   tp <- 6.280690
   err <- 0.029055
   i <- 4.000000
```

```
    lam_h_i <- 6.280690
  tp <- 6.340547
  err <- 0.009440
  i <- 5.000000
    lam_h_i <- 6.340547
  tp <- 6.359983
  err <- 0.003056
  i <- 6.000000
----------------------------------------

 srn_exrss(icupc98; BH):   6.50059657e-003

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

 srn_exrss(icupc98; BN):   3.03008702e-002

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

 srn_exrss(icupc98; ACh):   8.70770327e+000

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

 srn_exrss(icupc98; fnum)/srn_exrss(icupc98; ftput2):   4.21143605e-004
```

2.4.10 Testing while statement

Description

This example is used to test the syntax of **while**-statement.

SHARPE File — *srn/syntaxtest*

```
bind i1 1
bind a 2

while i1 ≤ 3
loop j1, 1, 3, 1
bind k1 1
while k1 ≤ 3
expr i1, j1, k1
bind k1 k1+1
end
end
bind i1 i1+1
if a > 1
loop l1, 1, 3, 1
expr l1
end
end
end

loop i2, 1, 3, 1
bind j2 1
while j2 ≤ 3
expr i2, j2
bind j2 j2+1
end
end

expr min(1, 2), max(1, 2)

echo ERROR: while cannot be used in func definition
```

```
func test ()
while a > 1
end
end

end
```

Chapter 3

Model Types Integrated

3.1 Phased-Mission Systems(PMS)

The PMS model is implemented by Xinyu Zang [18], which has the following features:

- An efficient BDD-based algorithm is used for analysis, where BDD stands for binary decision diagrams [8, 1].
- The system configuration in each phase is specified by a fault tree.
- Transient analysis is provided.

3.1.1 Specification of model

The paradigm of fault tree models is used to specify the system configuration in each phase. A PMS is specified as follows:

pms *name* { **(** *param_list* **)** }
<*phase_number phase_name duration*>
end

The *phase_number* specifies which phase the system configuration is in. The *phase_name* should be the same as the *system_name* in the fault tree in which the system configuration is specified. The *duration* specifies the duration of this phase.

3.1.2 System analysis function

The only system analysis function that can be used from PMS model is

tvalue(*t, system_name*)

that gives the unreliability of the PMS at time t. Note that there may be latent faults at the transition of phases. Two switch commands are used to set which time the **tvalue** uses:

- **ltimep**: set time as t_-, i.e. at the end of the phase $i-1$.
- **rtimep**: set time as t_+, i.e. at the beginning of the phase i.

There are two examples included in the next section.

3.1.3 Examples

A three-phase system

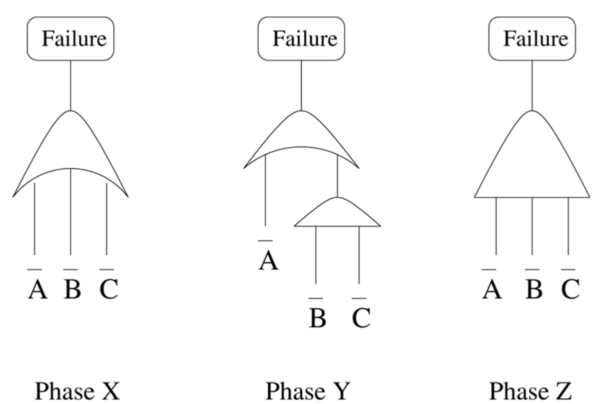


Figure 3.1: System configuration in three phases

Description The system has three phases X, Y and Z whose configurations are shown in Figure 3.1 in fault tree format. The equivalent system for the end of mission XYZ is

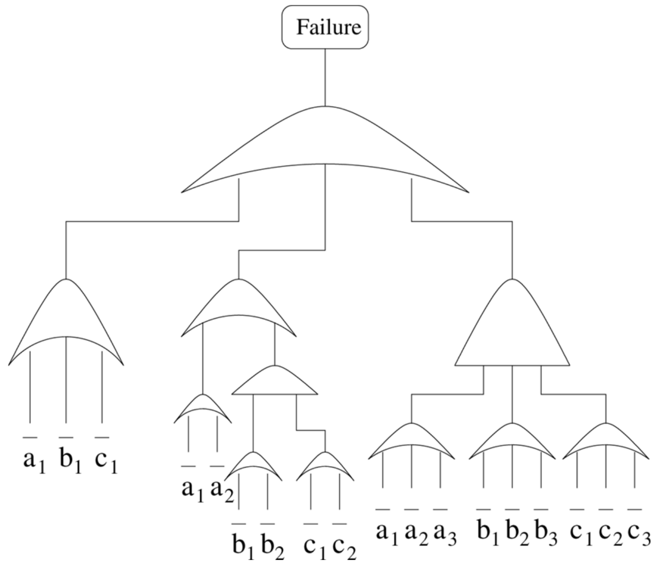


Figure 3.2: Equivalent system for the end of mission

shown in Figure 3.2. We also consider the other five possible phase configurations, i.e., XZY, YXZ, YZX, ZXY, ZYX.

SHARPE File — pms/yy.timep

```
format 8
epsilon results 0.000000000001

ftree X
basic a exp(a_x)
basic b exp(b_x)
basic c exp(c_x)
or top a b c
end

ftree Y
basic a exp(a_y)
```

```
basic b exp(b_y)
basic c exp(c_y)
and BC  b c
or top BC a
end

ftree Z
basic a exp(a_z)
basic b exp(b_z)
basic c exp(c_z)
and ABC a b c
end

bind
a_x    0.0001
a_y    0.0001
a_z    0.0001
b_x    0.0001
b_y    0.0001
b_z    0.0001
c_x    0.0001
c_y    0.0001
c_z    0.0001
T_x    10
T_y    10
T_z    10
end

pms XYZ
1  X  T_x
2  Y  T_y
3  Z  T_z
```

end

pms XZY

1 X T_x

2 Z T_z

3 Y T_y

end

pms YXZ

1 Y T_y

2 X T_x

3 Z T_z

end

pms YZX

1 Y T_y

2 Z T_z

3 X T_x

end

pms ZXY

1 Z T_z

2 X T_x

3 Y T_y

end

pms ZYX

1 Z T_z

2 Y T_y

3 X T_x

end

ltimep

```
loop t, 0, 30, 10
  expr tvalue(t; XYZ), tvalue(t; XZY)
  expr tvalue(t; YXZ), tvalue(t; YZX)
  expr tvalue(t; ZXY), tvalue(t; ZYX)
end

rtimep
loop t, 0, 30, 10
  expr tvalue(t; XYZ), tvalue(t; XZY)
  expr tvalue(t; YXZ), tvalue(t; YZX)
  expr tvalue(t; ZXY), tvalue(t; ZYX)
end

end
```

Space application

Description Modifying the space application in [11], we get an example whose mission alternates between operational phases *Launch*, *Asteroid*, *Comet*, with *Hibernation* phases as shown in Figure 3.3.

SHARPE File — pms/space

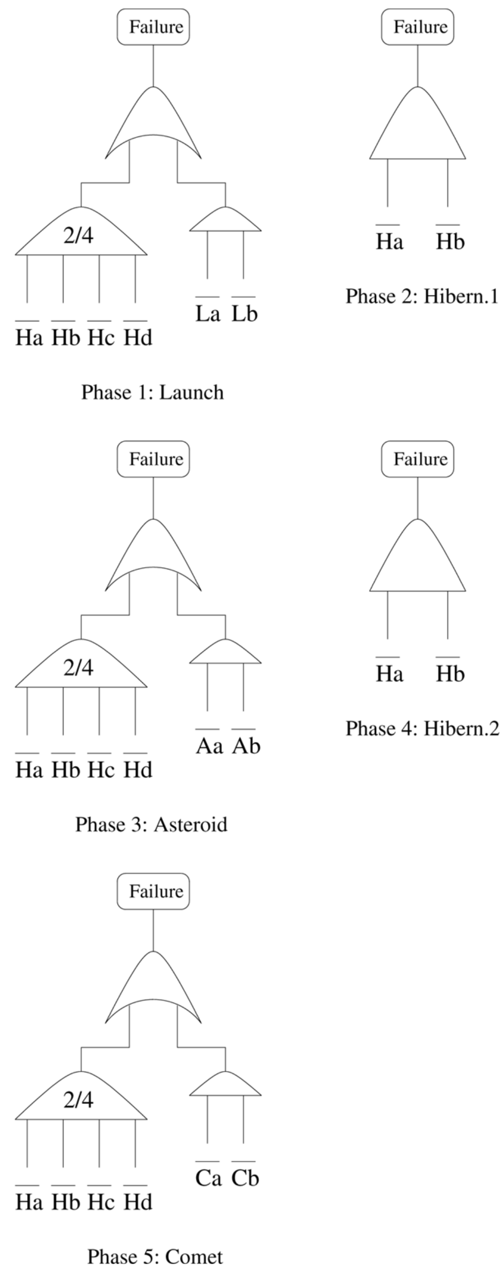


Figure 3.3: System configuration for space application

```
format 8

* Phase 1
ftree Launch
repeat    La    exp(RL)
repeat    Lb    exp(RL)
repeat    Ha    exp(RHo)
repeat    Hb    exp(RHo)
repeat    Hc    exp(RHo)
repeat    Hd    exp(RHo)
and     L       La Lb
kofn    H       2,4, Ha Hb Hc Hd
or    top    L H
end

* Phase 2
ftree Hibernation1
repeat    Ha    exp(RHh)
repeat    Hb    exp(RHh)
and    top    Ha Hb
end

* Phase 3
ftree Asteriod
repeat    Aa    exp(RA)
repeat    Ab    exp(RA)
repeat    Ha    exp(RHo)
repeat    Hb    exp(RHo)
repeat    Hc    exp(RHo)
repeat    Hd    exp(RHo)
and     A       Aa Ab
kofn    H       2,4, Ha Hb Hc Hd
```

```
or    top    A H
end

* Phase 4
ftree Hibernation2
repeat    Ha    exp(RHh)
repeat    Hb    exp(RHh)
and    top    Ha Hb
end

* Phase 5
ftree Comet
repeat    Ca    exp(RC)
repeat    Cb    exp(RC)
repeat    Ha    exp(RHo)
repeat    Hb    exp(RHo)
repeat    Hc    exp(RHo)
repeat    Hd    exp(RHo)
and    C      Ca Cb
kofn    H      2,4, Hd Hc Hb Ha
or    top    C H
end

bind
RL    0.00005
RA      0.00001
RC      0.0001
RHo      0.00001
RHh      0.000001
T1     48
T2     17520
T3     672
```

```
T4      26952
T5    672
end

pms Space
1    Launch        T1
2    Hibernation1    T2
3    Asteriod    T3
4    Hibernation2    T4
5    Comet        T5
end

loop t, T1+T2+T3+T4, T1+T2+T3+T4+T5, 112
expr tvalue(t; Space)
end

end
```

Result Unreliability of space application

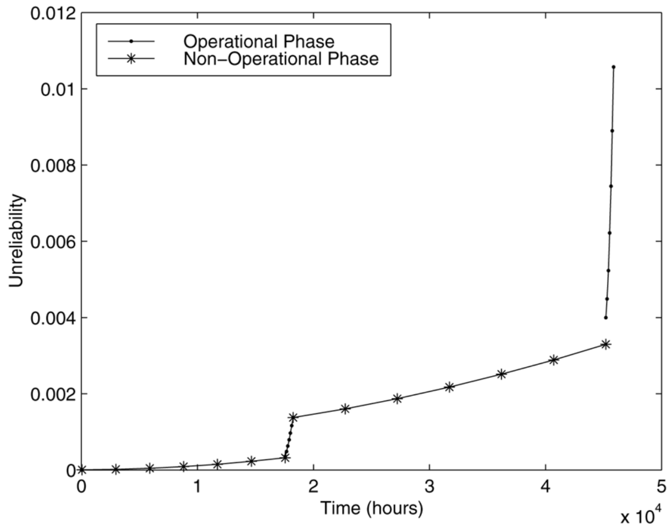


Figure 3.4: Unreliability of space application

3.2 Multistate Fault Trees

The Multi-state Fault Tree (MFT) model [18] is added in SHARPE as a new model that has the following features:

- An efficient BDD-based analysis algorithm is used for the MFT solution.
- The specification of MFT model is an extension of fault tree model.
- Most types of results for fault tree model are supported.

3.2.1 Specification of model

A multi-state fault tree is specified by the following:

> **mstree** *name* { **(** *param_list* **)** }
>
> <*mstreeline*>
>
> **end**

An *mstreeline* has one of the following forms:

1. **basic** *name:state ep*

   This is a basic component type. It is assigned a name, a state and an exponential polynomial. Whenever this name appears later in the multi-state tree specification, it is interpreted as being the same state of the same physical component.

2. **transfer** *name name*

   The second name must have been previously defined using **basic**. Whenever the first name appears later in the multi-state tree specification, it is interpreted as being the same physical component as the second name.

3. **and** *name name*{*:state*} *name*{*:state*} { *name*{*:state*} ... }

   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.

4. **or** *name name*{*:state*} *name*{*:state*} { *name*{*:state*} ... }

   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.

5. **kofn** *name expression, expression, name*{*:state*}

   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.

6. **kofn** *name expression, expression, name*{*:state*} *name*{*:state*} { *name*{*:state*} ... }

   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.

In forms 2 through 6, the names making up the block must already be defined. The block names that are *top:state* represent a state of top event in multi-state tree.

3.2.2 System analysis functions

Most types of results for fault tree model are supported, except for importance measure and mincuts. A state of top event (*top:state*) needs to be specified at *state_eword* in the corresponding functions. For example, if the **cdf** is asked for a state of top event, 1, in a multi-state tree, *mst*, **cdf**(*mst, top:1*) can give the result. Detailed description of fault tree models can be found in [14].

3.2.3 Examples

Two boards system

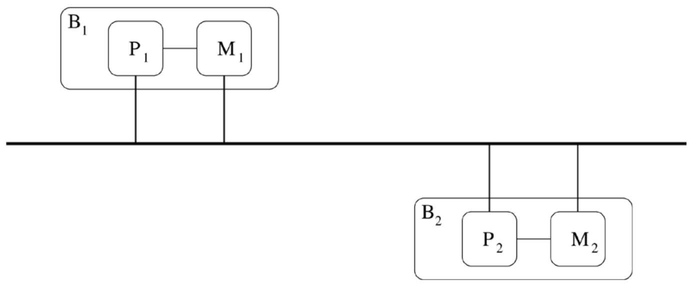


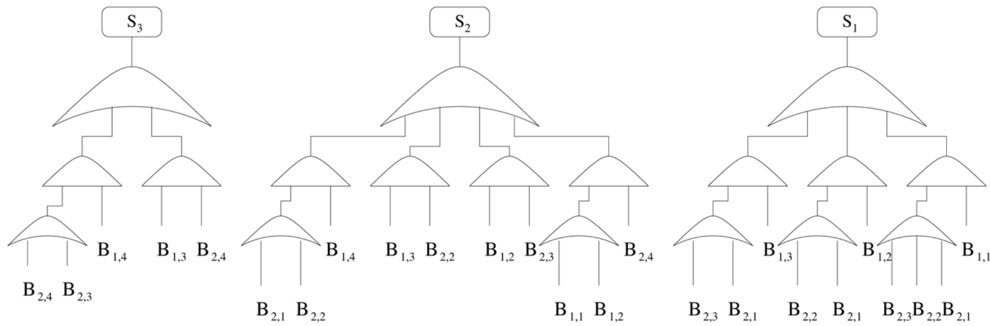
Figure 3.5: System diagram

Description Figure 3.5 shows a system with two boards B_1 and B_2, each having a processor and a memory. The memories (M_1 and M_2) can be shared by both processors (P_1 and P_2). The processor and memory on the same board can fail separately, but s-dependently. We define system state as: **state 1**, no processor or no memory are functional; **state 2**, at least one processor and exactly one memory are functional; **state 3**, at least one processor and both of the memories are functional. Figure 3.6 shows the MFTs for all the states of the system, where B_{ij} represents the board B_i being in **state j**.

SHARPE File — ms/ex1

```
format 8

mstree ex1
basic    B1:4    prob(0.95)
```



(a) MFT for system state 3 (b) MFT for system state 2 (c) MFT for system state 1

Figure 3.6: MFTs of example 3.2.3

basic B1:3 prob(0.02)

basic B1:2 prob(0.02)

basic B1:1 prob(0.01)

basic B2:4 prob(0.95)

basic B2:3 prob(0.02)

basic B2:2 prob(0.02)

basic B2:1 prob(0.01)

or gor321 B2:3 B2:4

and gand311 B1:4 gor321

and gand312 B1:3 B2:4

or top:3 gand311 gand312

or gor221 B1:1 B1:2

or gor222 B2:1 B2:2

and gand211 B1:4 gor222

and gand212 B1:3 B2:2

and gand213 B1:2 B2:3

and gand214 gor221 B2:4

or top:2 gand211 gand212 gand213 gand214

or gor121 B2:3 B2:1

or gor122 B2:2 B2:1

```
or    gor123    B2:3    B2:2    B2:1
and    gand111    B1:3    gor121
and    gand112    B1:2    gor122
and    gand113    B1:1    gor123
or    top:1    gand111    gand112    gand113
end

expr sysprob(ex1, top:1)
expr sysprob(ex1, top:2)
expr sysprob(ex1, top:3)

end
```

A communication network

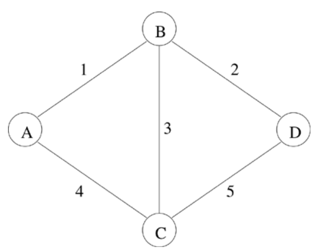


Figure 3.7: The network topology of example 3.2.3

Description Figure 3.7 shows a communication network topology. Each link can support c calls/connectionls simultaneously and the amount of bandwidth required by each call/connnection is equal, which means the call/connections are homogeneous. Obviously, the spare capacity of each link has multiple states: $0, 1, \ldots, c$. We assume the transitions among the states form a birth-death process with parameter λ and μ represented as a Con-

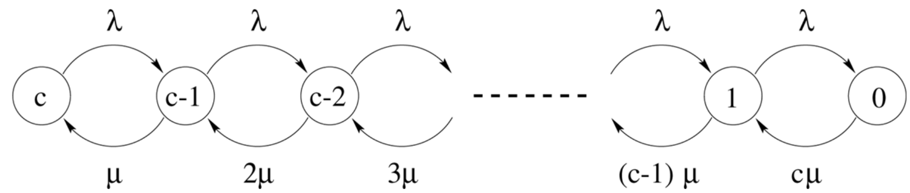


Figure 3.8: The CTMC for each link's spare capacity in example 3.2.3

tinuous Time Markov Chain (CTMC) in Figure 3.8. If there is an application which needs k simultaneous connections from A to D and all the k connections must follow the same route, we can obtain the blocking probability by MFT. The MFT is shown at Figure 3.9, and the blocking probability is $1 - P_S(t)$. Let c for all links be 10, and we calculate the blocking probability.

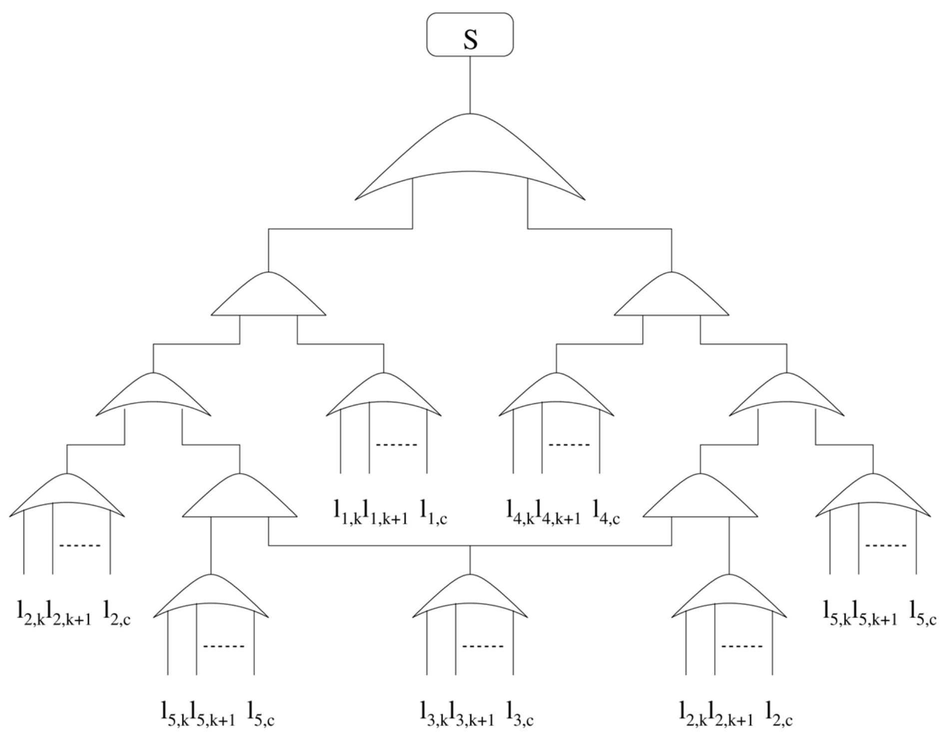


Figure 3.9: The MFT of example 3.2.3

SHARPE File — ms/app

```
format 8
epsilon results 0.000000000001

bind
lambda 0.1
mu 0.1
t 20000
end

markov link readprobs
10 9 lambda
9 8 lambda
8 7 lambda
7 6 lambda
6 5 lambda
5 4 lambda
4 3 lambda
3 2 lambda
2 1 lambda
1 0 lambda
0 1 10 * mu
1 2 9 * mu
2 3 8 * mu
3 4 7 * mu
4 5 6 * mu
5 6 5 * mu
6 7 4 * mu
7 8 3 * mu
8 9 2 * mu
9 10 mu
end
```

```
10    1.0
end


*debug mstree

mstree net(t)
basic    link1:0     prob(value(t; link, 0))
basic    link1:1     prob(value(t; link, 1))
basic    link1:2     prob(value(t; link, 2))
basic    link1:3     prob(value(t; link, 3))
basic    link1:4     prob(value(t; link, 4))
basic    link1:5     prob(value(t; link, 5))
basic    link1:6     prob(value(t; link, 6))
basic    link1:7     prob(value(t; link, 7))
basic    link1:8     prob(value(t; link, 8))
basic    link1:9     prob(value(t; link, 9))
basic    link1:10    prob(value(t; link, 10))
basic    link2:0     prob(value(t; link, 0))
basic    link2:1     prob(value(t; link, 1))
basic    link2:2     prob(value(t; link, 2))
basic    link2:3     prob(value(t; link, 3))
basic    link2:4     prob(value(t; link, 4))
basic    link2:5     prob(value(t; link, 5))
basic    link2:6     prob(value(t; link, 6))
basic    link2:7     prob(value(t; link, 7))
basic    link2:8     prob(value(t; link, 8))
basic    link2:9     prob(value(t; link, 9))
basic    link2:10    prob(value(t; link, 10))
basic    link3:0     prob(value(t; link, 0))
basic    link3:1     prob(value(t; link, 1))
basic    link3:2     prob(value(t; link, 2))
```

basic	link3:3	prob(value(t; link, 3))
basic	link3:4	prob(value(t; link, 4))
basic	link3:5	prob(value(t; link, 5))
basic	link3:6	prob(value(t; link, 6))
basic	link3:7	prob(value(t; link, 7))
basic	link3:8	prob(value(t; link, 8))
basic	link3:9	prob(value(t; link, 9))
basic	link3:10	prob(value(t; link, 10))
basic	link4:0	prob(value(t; link, 0))
basic	link4:1	prob(value(t; link, 1))
basic	link4:2	prob(value(t; link, 2))
basic	link4:3	prob(value(t; link, 3))
basic	link4:4	prob(value(t; link, 4))
basic	link4:5	prob(value(t; link, 5))
basic	link4:6	prob(value(t; link, 6))
basic	link4:7	prob(value(t; link, 7))
basic	link4:8	prob(value(t; link, 8))
basic	link4:9	prob(value(t; link, 9))
basic	link4:10	prob(value(t; link, 10))
basic	link5:0	prob(value(t; link, 0))
basic	link5:1	prob(value(t; link, 1))
basic	link5:2	prob(value(t; link, 2))
basic	link5:3	prob(value(t; link, 3))
basic	link5:4	prob(value(t; link, 4))
basic	link5:5	prob(value(t; link, 5))
basic	link5:6	prob(value(t; link, 6))
basic	link5:7	prob(value(t; link, 7))
basic	link5:8	prob(value(t; link, 8))
basic	link5:9	prob(value(t; link, 9))
basic	link5:10	prob(value(t; link, 10))

or slink1 link1:3 link1:4 link1:5 link1:6 link1:7 link1:8 link1:9 link1:10

or slink2 link2:3 link2:4 link2:5 link2:6 link2:7 link2:8 link2:9 link2:10

```
or    slink3    link3:3    link3:4    link3:5    link3:6    link3:7    link3:8    link3:9    link3:10
or    slink4    link4:3    link4:4    link4:5    link4:6    link4:7    link4:8    link4:9    link4:10
or    slink5    link5:3    link5:4    link5:5    link5:6    link5:7    link5:8    link5:9    link5:10
and    and4l    slink5    slink3
and    and4r    slink2    slink3
or    or3l    slink2    and4l
or    or3r    slink5    and4r
and    and2l    slink1    or3l
and    and2r    slink4    or3r
or    top:1    and2l    and2r
end



loop t, 5, 100, 5
expr 1−sysprob(net, top:1; t)
expr 1−value(t;link,10)−value(t;link,9)
expr 1−value(t;link,10)−value(t;link,9)−value(t;link,8)−value(t;link,7)
bind temp 1−value(t;link,10)−value(t;link,9)−value(t;link,8)−value(t;link,7)
expr temp−value(t;link,6)−value(t;link,5)
expr temp−value(t;link,6)−value(t;link,5)−value(t;link,4)−value(t;link,3)
end

end
```

Result Transient analysis of the application at $\lambda = 0.1$

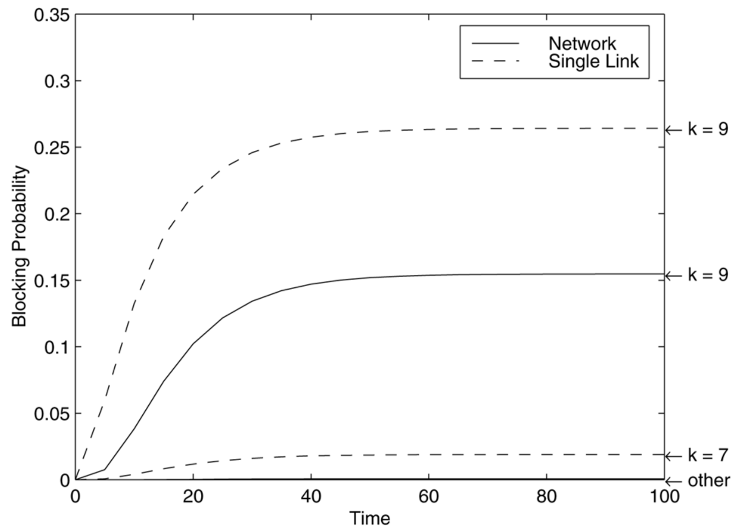


Figure 3.10: Transient analysis

3.3 Markov Regenerative Process [17]

3.3.1 Specification of model

mrgp *name* $\{$(*param_list*)$\}$

$*$ section 1: transitions and transition destributions

< *nodename1 edgetype nodename2 ep*>

$*$ section 2: rewards (optional)

$\{$**reward**

< *name expression*>$\}$

end

where *nodename1* is the starting node and *nodename2* is the destination node as in Markov and semi-Markov models, *edgetype* is either for Markov regenerative edges, or for non-regenerative edges, *ep* represents a distribution function, which could be **zero**, **inf**, **prob**(p),

exp(λ), **gen**, **cgen**, **tgen**, **cdf**, **Erlang** (n, λ), **hypoexp** (μ_1, μ_2), **hyperexp** (μ_1, p_1, μ_2, p_2), **mixture** (p_1, p_2, μ), **defective** (p, μ), **inst_unavail** (λ, μ), **ss_unavail** (λ, μ), **oneshot** (p), **activeE** (μ), **activeU** (μ_1, μ_2), **standbyE** (μ, μ_{sense}), **standbyU** $(\mu_1, \mu_2, \mu_{sense})$, **binomial** (λ, k, n), **kofn_ftree** (λ, k, n), **kofn_block** (λ, k, n), or any of user-defined distribution functions. Detailed description of the first 8 distribution functions can be found in Appendix B of [14].

3.3.2 System analysis functions

Only steady-state solution of MRGP models is given and the following functions are supported:

- **prob** (*sys_name*, *nodename* $\{$; *arglist*$\}$)

  Gets the steady state probability for node *nodename* of the MRGP model named *sys_name*.

- **exrss** (*sys_name*$\{$; *arglist*$\}$)

  Calculates the expected steady-state reward rate value.

3.3.3 Example – Cellular Networks with Generally Distributed Hand-off Traffic

Source

S. Dharmaraja, and K. Trivedi, Performance Analysis of Cellular Networks with Generally Distributed Hand-off Traffic, COMMUNICATED, 2001.

Description

Consider a single cell in a TDMA (Time Division Multiple Access) wireless system, where the base transceiver system of the cell has N base repeaters, one controller and a local area network connecting these subsystems. Each base repeater provides M time-division-multiplexed channels. The cell reserves one channel for signaling transfer (namely control channel), which resides in one of N base repeaters. Therefore, the total number of available channels for calls in the cell is $NM - 1\ (= C)$. For convenience in demonstrating the approach, we assume that the system has hexagonal geometry and that the cellular system is homogeneous. That is, all the cells are identical and have the same statistical behavior.

A call is accepted only when the cell can find a channel not in use, otherwise, the call is rejected. Call arrivals in cellular system can be classified as new calls and hand-off calls. New calls are generated by mobile originating or mobile terminating connections established in the initial cells, whereas hand-off calls are ongoing calls transferring from other cells. A hand-off call could fail due to insufficient bandwidth available in the new cell, and in such case, a drop of hand-off call occurs.

The dropping of a hand-off call is considered more severe than the blocking of a new call. One method ([7, 9]) to reduce the dropping probability of hand-off calls is to reserve a fixed number of channels exclusively for hand-off calls. These exclusively reserved channels are referred as *guard* channels. For example, if the total number of channels is C and the number of guard channels in the channel pool is g, then the number of available channels for new calls is $C - g$.

We assume that an ongoing call (new or hand-off) completion times are exponential with parameter μ_d and the time at which the mobile station engaged in the call departs the cell are exponential with parameter μ_h. We also assume that the inter-arrival times of hand-off calls are generally distributed with distribution function $G(t)$ and with finite mean $1/\lambda_h$ which is independent of new calls arrival time. Note that new calls who find all $C - g$ channels are busy leave the system whereas hand-off calls who find all C channels are busy leave the system. The state transition diagram for this model is shown in Figure 3.11.

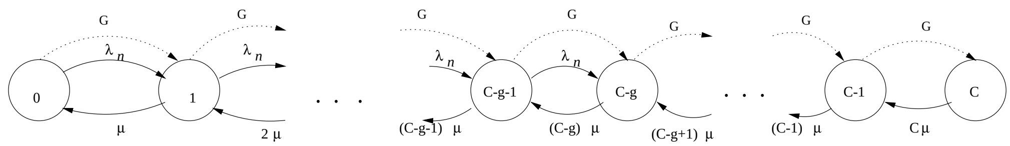


Figure 3.11: State transition diagram using MRGP modeling

SHARPE File — *mrgp/cellular*

```
format 8

bind
lambdaE 63
lambda  49
mu      1
end

* C = 5, g = 3

mrgp    cellular5_3
0 – 1   exp(lambda)
1 – 0   exp(mu)
1 – 2   exp(lambda)
2 – 1   exp(2*mu)
3 – 2   exp(3*mu)
4 – 3   exp(4*mu)
5 – 4   exp(5*mu)
0 @ 1   Erlang(3, lambdaE)
```

1 @ 2   Erlang(3, lambdaE)

2 @ 3   Erlang(3, lambdaE)

3 @ 4   Erlang(3, lambdaE)

4 @ 5   Erlang(3, lambdaE)

reward

2   1

3   1

4   1

5   1

end

∗ C = 6, g = 3

mrgp   cellular6_3

0 − 1   exp(lambda)

1 − 0   exp(mu)

1 − 2   exp(lambda)

2 − 1   exp(2∗mu)

2 − 3   exp(lambda)

3 − 2   exp(3∗mu)

4 − 3   exp(4∗mu)

5 − 4   exp(5∗mu)

6 − 5   exp(6∗mu)

0 @ 1   Erlang(3, lambdaE)

1 @ 2   Erlang(3, lambdaE)

2 @ 3   Erlang(3, lambdaE)

3 @ 4   Erlang(3, lambdaE)

4 @ 5   Erlang(3, lambdaE)

5 @ 6   Erlang(3, lambdaE)

reward

3   1

4   1

```
5      1
6      1
end

* C = 7, g = 3

mrgp   cellular7_3
0 - 1   exp(lambda)
1 - 0   exp(mu)
1 - 2   exp(lambda)
2 - 1   exp(2*mu)
2 - 3   exp(lambda)
3 - 2   exp(3*mu)
3 - 4   exp(lambda)
4 - 3   exp(4*mu)
5 - 4   exp(5*mu)
6 - 5   exp(6*mu)
7 - 6   exp(7*mu)
0 @ 1   Erlang(3, lambdaE)
1 @ 2   Erlang(3, lambdaE)
2 @ 3   Erlang(3, lambdaE)
3 @ 4   Erlang(3, lambdaE)
4 @ 5   Erlang(3, lambdaE)
5 @ 6   Erlang(3, lambdaE)
6 @ 7   Erlang(3, lambdaE)
reward
4      1
5      1
6      1
7      1
end
```

```
expr prob(cellular5_3, 5)
expr exrss(cellular5_3)
expr prob(cellular6_3, 6)
expr exrss(cellular6_3)
expr prob(cellular7_3, 7)
expr exrss(cellular7_3)

end
```

3.4 Reliability Block Diagrams

3.4.1 Specification of model [14]

A reliability block diagram is specified by:

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

An *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. There must be at least two components.

3. **or** *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. There must be at least two components.

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

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

5. **kofn** *name expression, expression, name name* $\{$ *name* ... $\}$

   This represents a *k*-out-of-*n* system whose components need not be identical. The system is assigned the first name. The first expression gives *k* and the second expression gives *n*. The names following the second expression are the components to the system; there must be at least two.

Detailed description of how to analyze reliability block diagrams can be found in Appendix B of [14].

3.4.2 Example – 2 Processors, 3 Memories System

Description

A system has 2 processors and 3 Memories. Each processor has a failure rate λ_p. Each memory has a failure rate λ_m. The system is up if at least one processor and at least k (1 or 2) memories are up. The reliability block diagram for $k = 1$ is shown in Figure 3.12.

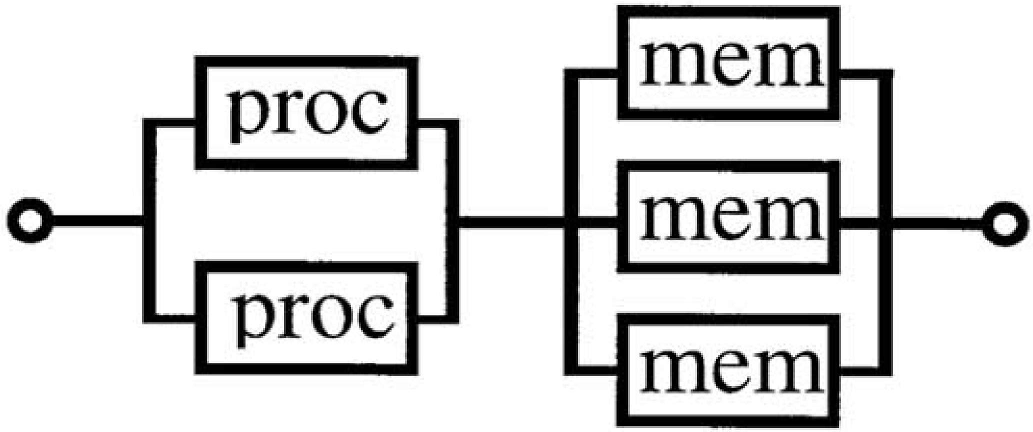


Figure 3.12: Reliability block diagram for the 2 processors, 3 memories system

SHARPE File — *block/2p3m.block*

```
* Two−processors, three−memories system
* Use a block diagram to model system reliability
* k is the minimum number of memories needed

format 8

block nodep(k)
comp proc exp(lambdap)
comp mem  exp(lambdam)
parallel procs proc proc
kofn mems k,3,mem
series top procs mems
end

* Now assign failure rate values
```

```
bind
lambdap 1/720
lambdam 1/(2*720)
end

* Compare mean time to system failure under
* two conditions: a minimum of
* one memory required vs. 2 memories
* find the difference between the use of tvalue and value
expr mean(nodep;1), mean(nodep;2), mean(nodep;1)/mean(nodep;2)

* Now compare system unreliabilities
func unrel1(t) tvalue(t;nodep;1)
func unrel2(t) tvalue(t;nodep;2)
loop t,0,50,10
expr unrel1(t), unrel2(t)
end

end
```

3.5 Fault Trees

3.5.1 Specification of model

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 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 an 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. **not** *name name*

   This represents a "not" gate. The gate output is assigned the first name, and the second names form the input to the gate. See the example C.1.2.

4. **transfer** *name name*

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

5. **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.

6. **nand** *name name name* { *name* ... }

   This represents a "nand" gate. The gate output is assigned the first name, and the rest of the names form the inputs to the gate. There must be at least two inputs. See the example C.1.1.

7. **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.

8. **nor** *name name name* { *name* ... }

   This represents a "nor" gate. The gate is output assigned the first name, and the rest of the names form the inputs to the gate. There must be at least two inputs. See the example C.1.1.

9. **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.

10. **nkofn** *name expression, expression, name*

    This represents a *not* k-out-of-n gate having identical inputs. The gate output 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.

11. **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.

12. **nkofn** *name expression, expression, name name* { *name* ... }

    This represents a *not* 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. See the example C.1.2.

In forms 2 through 8, the names making up the block must already be defined.

3.5.2 System analysis functions

New analysis functions and new features are listed as the following. Other analysis functions are described in Appendix B of [14].

1. **mincuts**(*system_name* {; *arglist*})

   This prints out the set of mincuts of a fault tree (See the example C.1.3).

2. Results for gate:

   User can obtain results at each gate output by assigning the name of the gate to *state_eword* in corresponding function. For example, if the **cdf** is asked for gate, *gn*, in a fault tree, *ft*, **cdf**(*ft*, *gn*) can give the result.

3. Importance measure for an event:

   Three types of importance measure can be obtained from a fault tree model (see the example C.1.4):

   (a) **bimpt**(t; *system_name, event_name* {; *arglist*})

   This gives Birnbaum's importance for event, *event_name,* at time t.

   (b) **cimpt**(t; *system_name, event_name* {; *arglist*})

   This gives criticality importance for event, *event_name*, at time t.

   (c) **simpt**(*system_name, event_name* {; *arglist*})

   This gives structural importance for event, *event_name*.

3.5.3 Example – 2 Processors, 3 Memories System

Description

This is the same system introduced in chapter 3.4.2. The corresponding fault tree is in Figure 3.13, where $P1$ and $P2$ represent the two processors, and $M1$, $M2$, and $M3$ denote the three memories, respectively. Furthermore, μ_p and μ_m have been introduced as independent repair rates for each processor and each memory, respectively. Then, the instantaneous unavailability of the system has been calculated via the model named *indrep* in the SHARPE file listed at the chapter 3.5.3.

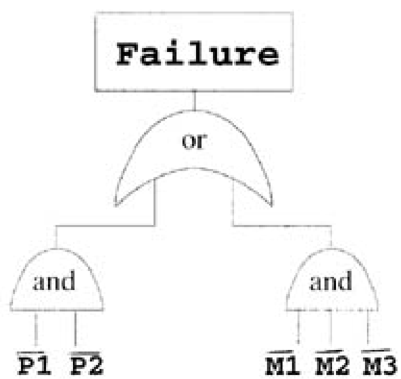


Figure 3.13: Fault tree for the 2 processors, 3 memories system

SHARPE File — *ftree/2p3m.ftree*

∗ 2 processors, 3 memories system modeled by fault tree

format 8

```
ftree nodepf(k)
basic proc exp(lambdap)
basic mem exp(lambdam)
and procs proc proc
kofn mems (4−k),3,mem
or top procs mems
end

* Now assign failure rate values
bind
lambdap 1/720
lambdam 1/(2*720)
end

* note the difference in kofn of ftree with block
* Compare answers obtained by two
* distinct models of the same system
expr mean(nodepf;1), mean(nodepf;2), mean(nodepf;1)/mean(nodepf;2)

* Assume Independent Failure And Independent Repair
* model system insta. availability
ftree indrep(k)
basic proc inst_unavail(lambdap,mup)
basic mem inst_unavail(lambdam,mum)
and procs proc proc
kofn mems (4−k),3,mem
or top procs mems
end

* Assign Repair Rate Values
bind
mup 1/2.5
```

```
mum 1/2.5
end
* Now compare system unavailabilities
func unavail1(t) tvalue(t;indrep;1)
func unavail2(t) tvalue(t;indrep;2)
loop t,0,50,10
expr unavail1(t), unavail2(t)
end

end
```

3.6 Reliability Graphs

3.6.1 Specification of model

A reliability graph is specified by the following:

> **relgraph** *name* { (*param_list*) }
>
> ∗ section 1: unidirectional edges
>
> <*edge_name edge_name ep* { **transfer** *edge1_name edge1_name*{ *edge2_name*
> *edge2_name* . . . }}>
>
> ∗ section 2:bidirectional edges (optional)
>
> { **bidirect**
>
> <*edge_name edge_name ep* { **transfer** *edge1_name edge1_name*{ *edge2_name*
> *edge2_name* . . . }}>}
>
> **end**

The **transfer** part in the above specification is the extension that defines the repeated edges. The *edge1* from the first *edge1_name* to the second *edge1_name* is repeated for the *edge* from the first *edge_name* to the second *edge_name*. So are the optional edges from the fist *edge***i***_name* to the second *edge***i***_name*. Examples of repeated edges are listed in chapter 3.6.3.

3.6.2 System analysis functions

Two new types of system analysis functions are integrated as the following (for others, see Appendix B of [14]):

1. Mincuts and minpaths set:

   (a) **mincuts**(*system_name* {; *arglist*})

   This prints out the set of mincuts of a reliability graph. See the example C.2.1.

   (b) **minpaths**(*system_name* {; *arglist*})

   This prints out the set of minpaths of a reliability graph. See the example C.2.2.

2. Importance measure for an edge:

   Three types of importance measure can be obtained from a reliability graph model (see the example C.2.3):

   (a) **bimpt**(t; *system_name, node_name, node_name* {; *arglist*})

   This gives Birnbaum's importance for edge, (*node_name, node_name*), at time t.

   (b) **cimpt**(t; *system_name, node_name, node_name* {; *arglist*})

   This gives criticality importance for edge, (*node_name, node_name*), at time t.

   (c) **simpt**(*system_name, node_name, node_name* {; *arglist*})

   This gives structural importance for edge, (*node_name, node_name*).

3.6.3 Examples

2 Processors, 3 Memories System with Inter-connection Dependence

Description This is still a system with 2 processors and 3 memories. Compared to the system mentioned in chapter 3.4.2 and chapter 3.5.3, inter-connection dependence has been considered. Processor $P1$ only uses memory $M1$ and $M3$, and processor $P2$ only uses memory $M2$ and $M3$. The system is up when at least one processor and one memory are working. In the following SHARPE file, the model rel_proc_mem2 is based on repeated edges. The reliability graph for the model rel_proc_mem is shown in Figure 3.14.

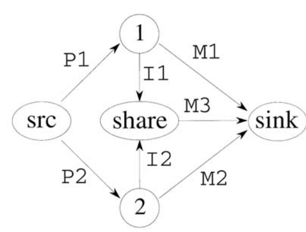


Figure 3.14: Reliability graph for the 2 processors, 3 memories system with inter-connection dependence without repeated edges

SHARPE File — *relgraph/repeat.txt*

∗ reliability graph for

∗ 2−processor,

∗ 3−memory system

relgraph rel_proc_mem

src P1 exp(1/Ptime)

src P2 exp(1/Ptime)

P1 sink exp(1/Mtime)

P2 sink exp(1/Mtime)

```
P1 share inf
P2 share inf
share sink exp(1/Mtime)
end

bdd on

relgraph rel_proc_mem2
src P1 exp(1/Ptime)
src P2 exp(1/Ptime)
P1 sink exp(1/Mtime)
P2 sink exp(1/Mtime)
P1 sink exp(1/Mtime) transfer P2 sink
end

bind
Ptime 720
Mtime 2*720
end

pqcdf(rel_proc_mem)
cdf(rel_proc_mem)

pqcdf(rel_proc_mem2)
cdf(rel_proc_mem2)

end
```

An Electrical-pyrotechnic System

Source A. Birolini, *Quality and Reliability of Technical Systems*, Springer-Verlag, Berlin Heidelberg, New York, 1994.

Description To separate a satellite's protective shielding, a special electrical-pyrotechnic system shown in Figure 3.15 is used. An electrical signal comes through the cables E_1 and E_2 (redundancy) to the electrical-pyrotechnic signal to explosive charges for guillotining bolts E_{12} and E_{13} of the tensioning belt. The charges can be ignited from two sides, although one ignition will suffice (redundancy). For fulfillment of the required function, both bolts must be exploded simultaneously. Calculate the probability of failure of this separation system.

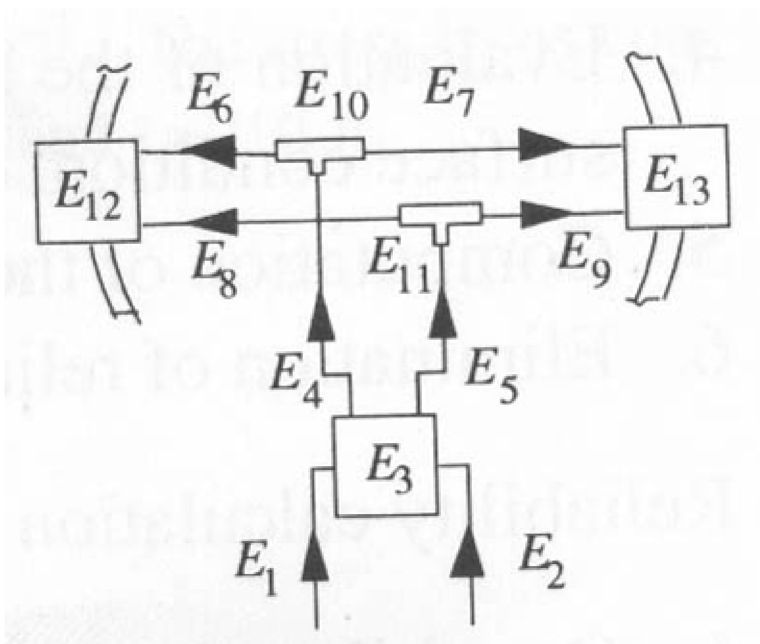


Figure 3.15: A special electrical-pyrotechnic system

SHARPE File — *relgraph/ex2.15*

```
relgraph ex2.15(e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12, e13)
src p1 exp(e1)
src p1 exp(e2)
p1 p2 exp(e3)
```

```
p2 p4 exp(e4) transfer p8 p10
p2 p3 exp(e5) transfer p8 p9
p6 p7 exp(e6)
p12 p13 exp(e7)
p5 p7 exp(e8)
p11 p13 exp(e9)
p4 p6 exp(e10) transfer p10 p12
p3 p5 exp(e11) transfer p9 p11
p7 p8 exp(e12)
p12 sink exp(e13)
end

pqcdf(ex2.15; 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

end
```

3.7 Series-parallel Acyclic Directed Graphs

3.7.1 Specification of model

A series-parallel graph is specified as follows:

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

A *graphline* has one of the following forms:

1. **dist** *name ep*

   this assigns the given *ep*, which is a defined distribution function, 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 on 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 an 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.

Detailed description of how to analyze series-parallel acyclic directed graph can be found in Appendix B of [14].

3.7.2 Example – A CPU-Input/Output Overlap System

Source

D.F. Towsley, J.C. Browne and K.M. Chandy, Models for Parallel Processing within Programs, *CACM*, October, 1978.

Description

Figure 3.16 shows a series-parallel graph representing one iteration of the program with CPU-Input/Output Overlap. In each iteration of the program, there are two stages. The first stage is always a CPU burst. The second stage consists of either pure I/O, or I/O that may be overlapped with a second CPU burst. As in Figure 3.16, the probability that the second stage contains CPU-I/O overlap is given by p. In the following SHARPE file, the model *OVERLAP* represents the model in Figure 3.16, while the model *SERIAL* denotes the model without CPU-I/O overlap. The speedup for various values of p has been computed.

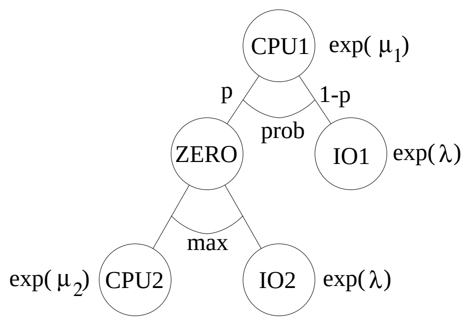


Figure 3.16: Precedence graph for the CPU-I/O overlap system

SHARPE File — *th/24*

```
*    CPU−I/O overlap

bind
mu1     1 / 0.0376
mu2     1 / 0.125
lambda  1 / 0.14995
end

graph  SERIAL(p)
```

```
cpu1    cpu2
cpu2    io2
cpu1    io1
end

exit    cpu1    prob
prob    cpu1    cpu2  p
dist    cpu1    exp ( mu1)
dist    io1     exp ( lambda)
dist    cpu2    exp ( mu2)
dist    io2     exp ( lambda)
end

graph  OVERLAP(p)
cpu1      zero1
cpu1       io1
zero1      cpu2
zero1       io2
end

exit    cpu1    prob
prob    cpu1    zero1  p
exit    zero1    max
dist    cpu1    exp ( mu1)
dist    zero1    zero
dist    io1     exp ( lambda)
dist    cpu2    exp ( mu2)
dist    io2     exp ( lambda)
end

expr mean(SERIAL;0.7)
expr mean(OVERLAP;0.7)
```

```
expr mean(SERIAL;0.6)/mean(OVERLAP;0.6)
expr mean(SERIAL;0.7)/mean(OVERLAP;0.7)
expr mean(SERIAL;0.8)/mean(OVERLAP;0.8)
expr mean(SERIAL;0.9)/mean(OVERLAP;0.9)
expr mean(SERIAL;1.0)/mean(OVERLAP;1.0)

bind
mu1    1 / 0.01
end

expr mean(SERIAL;1.0)/mean(OVERLAP;1.0)
end
```

3.8 Single-chain Product-form Queueing Networks

3.8.1 Specification of model

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
>
> <*stationline*>
>
> **end**
>
> section 3: number of customers per chain
>
> <*chain_name expression*>
>
> **end**

An *blockline* has one of the following forms:

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 once 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 has 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 numbers of jobs for which no rate was explicitly given.

Detailed explanation of how to analyze single-chain product-form queueing networks can be found in Appendix B of [14].

3.8.2 Example — a Terminal-oriented System with a Limited Number of Memory Partitions [16]

Description

This is the example 9.16 in [16]. As shown in Figure 3.17, the system has M terminals. Only n active jobs can concurrently share the main memory, which means $M = n$. Also, there is an assumption that the main memory is large enough so that no waiting in the job queue is required, which means the station *term* is an infinite server with the key word **is** assigned to it as mentioned in the previous section. The model tested in the following SHARPE file has $m = 3$.

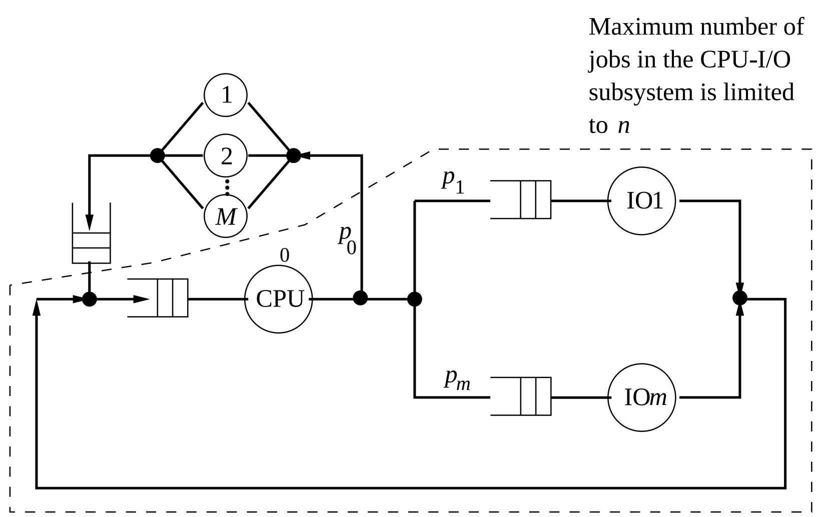


Figure 3.17: a Terminal-oriented System with a Limited Number of Memory Partitions

SHARPE File — *pfqn/9.16-nocon*

∗ This example is Ex 9.16 from the book.

∗ This implements the queueing network ignoring the

∗ memory constraint. This corresponds to E[Rˆ] in table 9.12

```
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 ER(M) M/ET(M) − 1/sterm
expr ER(10)
expr ER(20)
expr ER(30)
expr ER(40)
expr ER(50)
expr ER(60)
end
```

3.9 Multiple-chain Product-form Queueing Networks

3.9.1 Specification of model

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
>
> <*stationline*>
>
> { <*chain_name expression*, . . .> }
>
> **end**>
>
> **end**

∗ section 3: number of customers per chain

<chain_name expression>

end

Detailed explanation of how to analyze multiple-chain product-form queueing networks can be found in Appendix B of [14].

3.9.2 Example — a Terminal-oriented System with a Limited Number of Memory Partitions [16]

Description

This is the multiple-chain product-form queueing network version of the system mentioned in chapter 3.8.2.

SHARPE File — *mpfqn/inp9.16b*

```
* This example is Ex 9.16 from the book.  This implements the queueing
* network ignoring the
* memory constraint.  This corresponds to E[R^] in table 9.12

* results should be the same as for pfqn/9.16-nocon

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
mpfqn ex9.16(n)
chain cust
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
end
cpu fcfs scpu
end
term is sterm
end
io1 fcfs sio1
end
io2 fcfs sio2
end
io3 fcfs sio3
end
end
cust n
end
func ET(N) scpu*mutil(ex9.16,cpu;N)*p0
func ER(M) M/ET(M) − 1/sterm
expr ER(10)
```

expr ER(20)

expr ER(30)

expr ER(40)

expr ER(50)

expr ER(60)

end

3.10 Markov Chains

3.10.1 Specification of model

A Markov chain is specified as follows:

markov *name* {(*param_list*)} { **readprobs** }

∗ section 1: transitions and transition destributions

<*markov_edgeline*>

∗ section 2: rewards (optional)

{**reward** { **default** *expression*}

<*markov_setline*}>}

end

∗ section 3: initial state probabilities

{<*markov_setline*>}

end

{ **fastmttf**

< *name* **reada** >

< *name* **readf** >

end }

where *markov_edgeline* are either

name name expression

or

loop *simple_var*, *low*, *high* {,*increment* }
<*markov_edgeline*>
end

and *markov_setline* are either

name expression

or

loop *simple_var*, *low*, *high* {,*increment* }
<*markov_setline*>
end

which you can set reward rate or initial values to the node *name*.

Normally, an irreducible Markov chain doesn't have been specified with initial state probabilities, which means it is not necessary for an irreducible Markov chain to have section 3 unless users specify **readprobs**. Also, without initial state probabilities, **tvalue** and **prob** cannot be applied to irreducible Markov chains.

Fast mean time to failure(MTTF) is introduced from the paper [6] and requires the operating system running SHARPE supports IEEE 754 floating point standard. See the example at chapter C.3.1.

Detailed information of how to analyze Markov chains can be found in Appendix B of [14].

3.10.2 Example — Erlang Loss Model

Description

Consider a telephone switching system having n trunks with an infinite caller population. The arrival times are exponentially distributed with rate λ and call holding times are exponentially distributed with average $\frac{1}{\mu}$. When an arriving call finds all n trunks are busy, it is lost without further trying. Given number of non-failed channels, the principal quantity of interest is the *blocking probability*, which is obtained by the steady-state probability that all trunks are busy. The state diagram is shown in Figure 3.18.

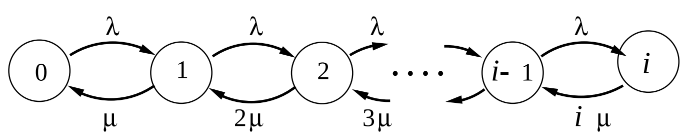


Figure 3.18: State diagram for the Erlang loss performance model

Assume that a single repair unit is shared by all the trunks. Also assume that the times to trunks failures and repair are exponentially distributed with rate γ and τ, respectively. The availability model is the CTMC in Figure 3.19.

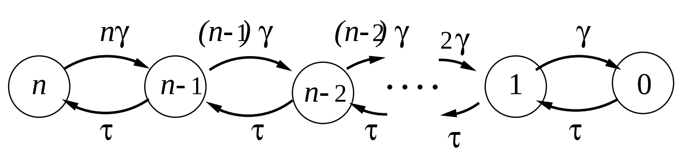


Figure 3.19: State diagram for the Erlang loss availability model

The composite model is shown in Figure 3.20. The state (i, j) represents that i non-failed trunks and j calls are currently in the system.

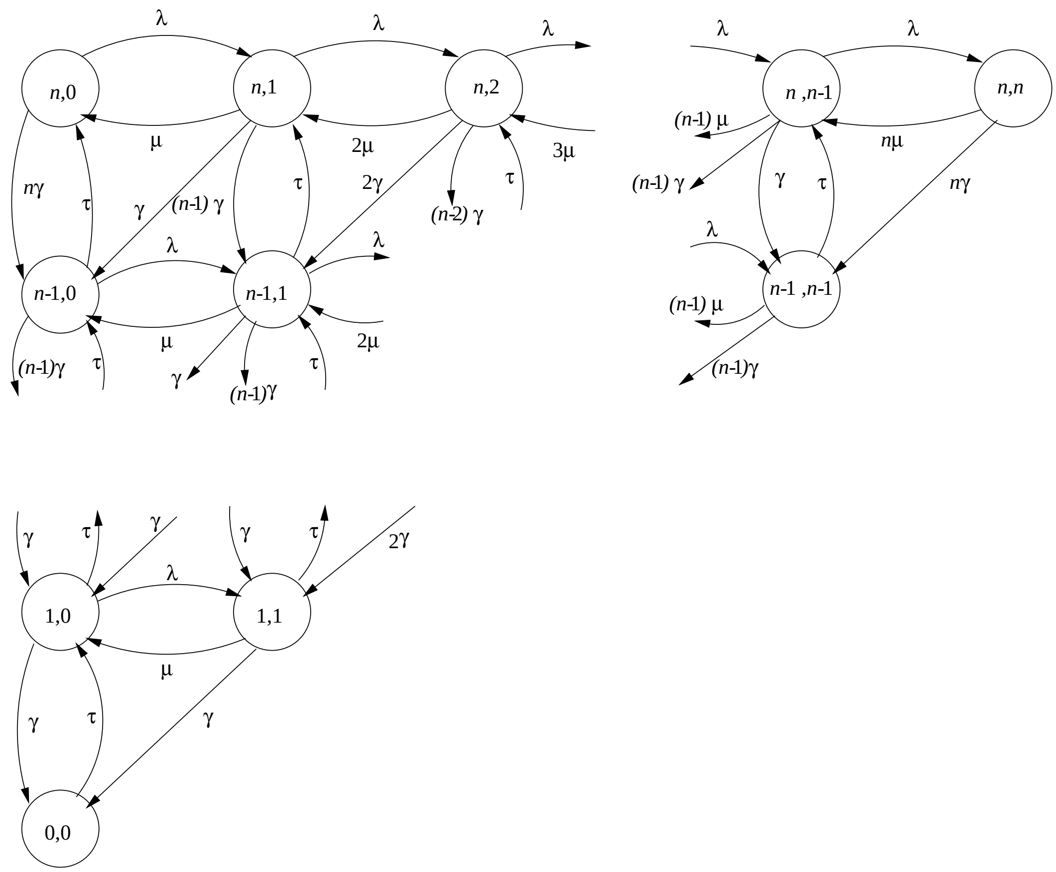


Figure 3.20: State diagram for the Erlang loss performability composite model

SHARPE File — *bluebook/8.27*

```
* This example is Ex 8.27 from the book.
* This implements the Erlang loss model.

format 8

bind
 lambda 49
 mu     3
```

```
MTTF 1000
  MTTR 24
end

* Hierarchical Model

* Availability submodel
markov perf(C)
  loop i,0,C-1
    $(i) $(i+1) lambda
    $(i+1) $(i) (i+1)*mu
  end
end
end

* function to use to define the reward rates for the measure
* the total call blocking  probability
* Reward function used for k>g
func Rew(C) prob(perf,$(C);C)

markov hier
loop i,C,1,-1
  $(i) $(i-1) i/MTTF
  $(i-1) $(i) 1/MTTR
end
reward
0 1
  loop i,1,C
     $(i) Rew(i)
  end
end
* Initial probability
```

```
$(C)  1
end

loop nb,35,45,1
  bind C nb
expr exrss(hier)
end

var Td exrss(hier)
loop nb,35,45,1
  bind C nb
  expr Td
end

* Composite model
markov cp
  loop j,1,C,1
    * Definition of the Availability part of the model
  * Downwards failure
    loop i,C,j,−1
      $(i)_$(j−1) $(i−1)_$(j−1) (i−j+1)/MTTF
      $(i−1)_$(j−1) $(i)_$(j−1) 1/MTTR
       * Definition of the Performance part of the model
      $(i)_$(j−1) $(i)_$(j) lambda
       $(i)_$(j) $(i)_$(j−1) j*mu
      * Diagonal failure
       $(i)_$(j) $(i−1)_$(j−1) (j)/MTTF
    end
  end
end
end
```

```
∗ Outputs
∗ Total call blocking  probability
var Tb sum(i,0,C, prob(cp,$(i)_$(i)))
var Unavail prob(cp,0_0)

loop nb,35,45,1
  bind C nb
  expr Tb
end

end
```

Result

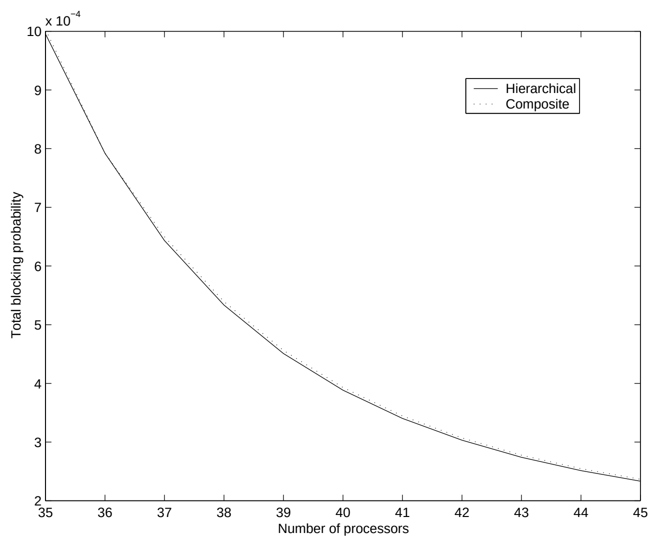


Figure 3.21: Total blocking probability in the Erlang loss performability model

3.11 Semi-Markov Chains

3.11.1 Specification of model

A semi-Markov chain is specified as follows:

semimark *name* {(*param_list*)} { **cond** | **uncond** }

∗ section 1: transitions and transition destributions

<*nodename1 nodename2 ep*>

∗ section 2: rewards (optional)

{**reward** { **default** *expression*}

<*name expression*>}

end

section 3: initial state probabilities

{<*name expression*>}

end

{ **fastmttf**

< *name* **reada** >

< *name* **readf** >

end }

An irreducible semi-Markov chain doesn't have section 3. The key word **fastmttf** is used for fast MTTF [6]. See the example C.3.2.

Detailed information of how to analyze semi-Markov chains can be found in Appendix B of [14].

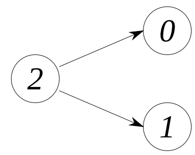


Figure 3.22: A semi-Markov chain

3.11.2 Example — Figure 3.22

SHARPE File — *semimark/1*

```
semimark main
2 1 gen\
  1, 0, 0\
  −1, 0, −lambda\
  −lambda, 1, −lambda
2 0 exp (.01)
end
end

bind
lambda .02
end

lcdf (main,2)
cdf (main,1)
cdf (main,0)
end
```

3.12 Generalized Stochastic Petri Nets

3.12.1 Specification of model

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, types 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

∗ section6: inhibitor arcs and multiplicity

<*place_name transition_name expression*>

end

Detailed information of how to analyze generalized Stochastic Petri Nets can be found in Appendix B of [14].

3.12.2 Example — M/M/1/K Queue with Server Failure and Repair

Description

The system has 1 server with buffer length K. So K jobs can be in the system at a time. The exponentially failure and repair rates for the server are γ and τ, respectively. See the Figure 3.23.

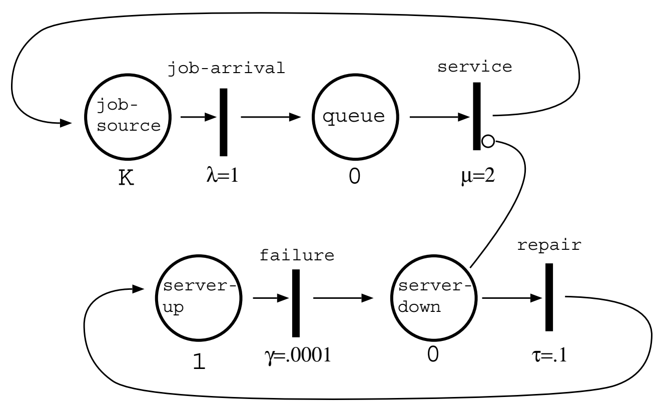


Figure 3.23: GSPN model for queue with server failure and repair

SHARPE File — *whitebook/mm1k.gspn*

∗ Initialize Variables

bind

LAM 1

MU 2

GAM 0.0001

TAU 0.1

```
inhibtok 1
end

gspn mm1k(K)
∗ Initial # of Tokens in Places
jobsource K
queue 0
serverup  1
serverdown  0
end
∗ Rates of Timed Transitions
jobarrival  ind LAM
service  ind MU
failure  ind GAM
repair  ind TAU
end
∗ No Immediate Transitions
end
∗ Input Arcs
jobsource jobarrival 1
queue service 1
serverup  failure 1
serverdown  repair 1
end
∗ Output Arcs
jobarrival  queue 1
service  jobsource 1
failure  serverdown  1
repair  serverup  1
end
∗ Inhibit Arcs
serverdown  service  inhibtok
```

```
end

var Lreject LAM*prempty(mm1k,jobsource;10)
var Pidle  prempty(mm1k,queue;10)
var Preject prempty(mm1k,jobsource;10)
var avquelength etok(mm1k, queue; 10)
var thruput tput(mm1k, service; 10)
var utilization util(mm1k, service; 10)

expr Pidle
expr Lreject, Preject
expr avquelength
expr thruput, utilization
end
```

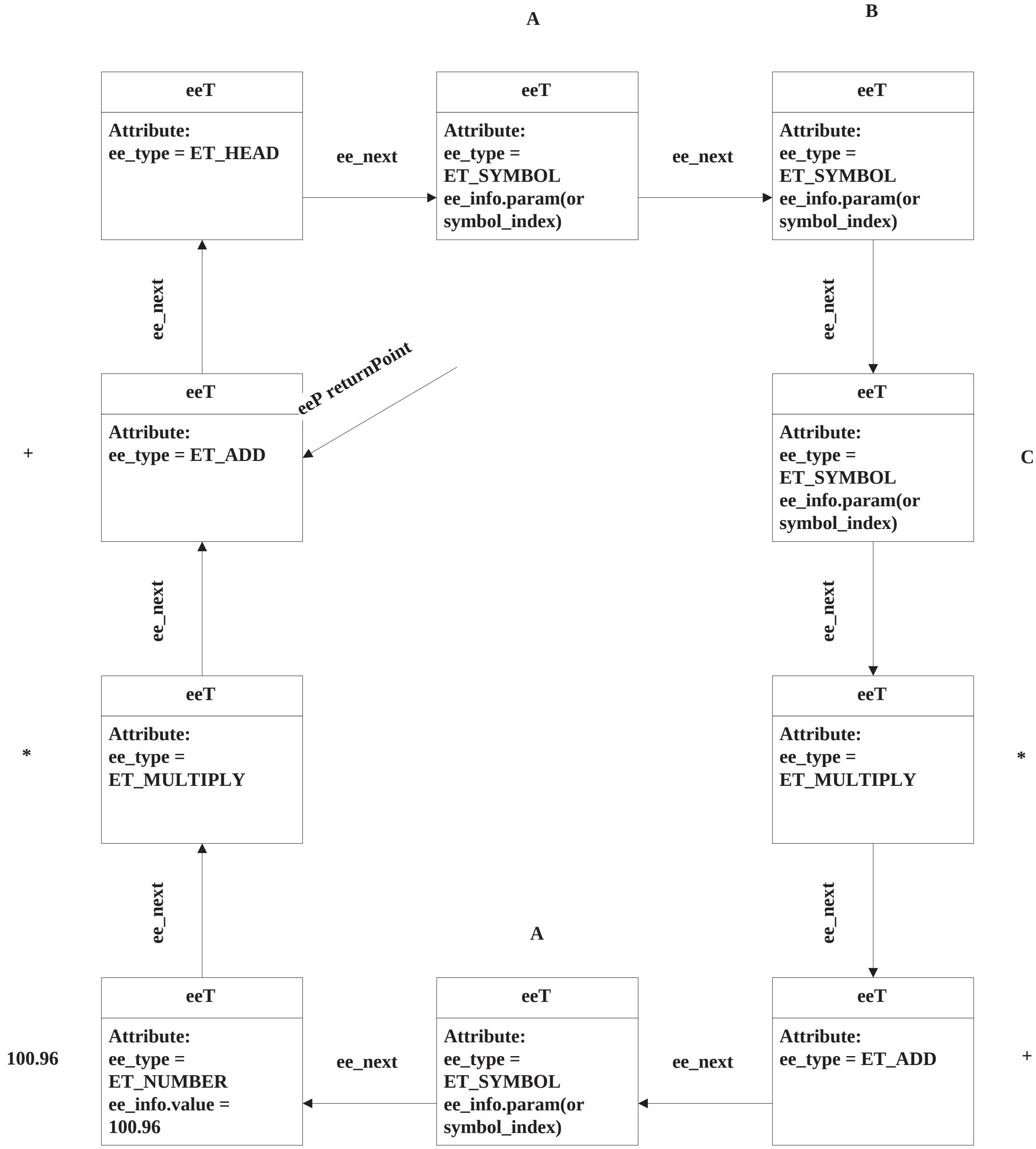
Appendix A

SHARPE Data Structure

Important data structures of SHARPE source code are listed here. Rectangles represent instances of data types with the name of each data type at the top of rectangles. These data types are $structure$s or $union$s in C language. For the sake of saving space, only important member field(s) are listed at the **attribute** field of each rectangle. Arcs represent pointers. If an arc begins from a rectangle, it is a field of the data type that the rectangle represents. Rectangles are piled together to denote $array$s in C.

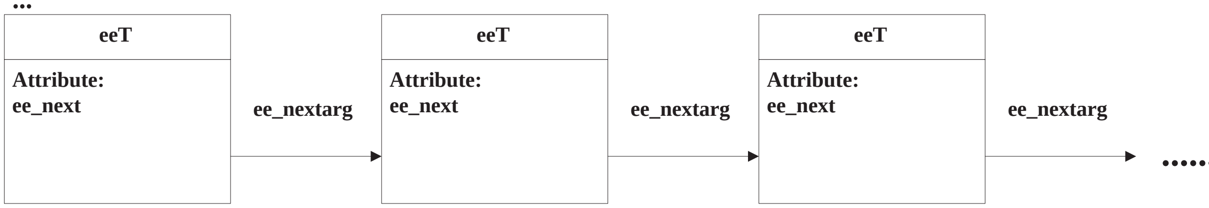
Basic EXPRESSION Sample

A + B * C + A * 100.96 stored as A B C * + A 100.96 * +

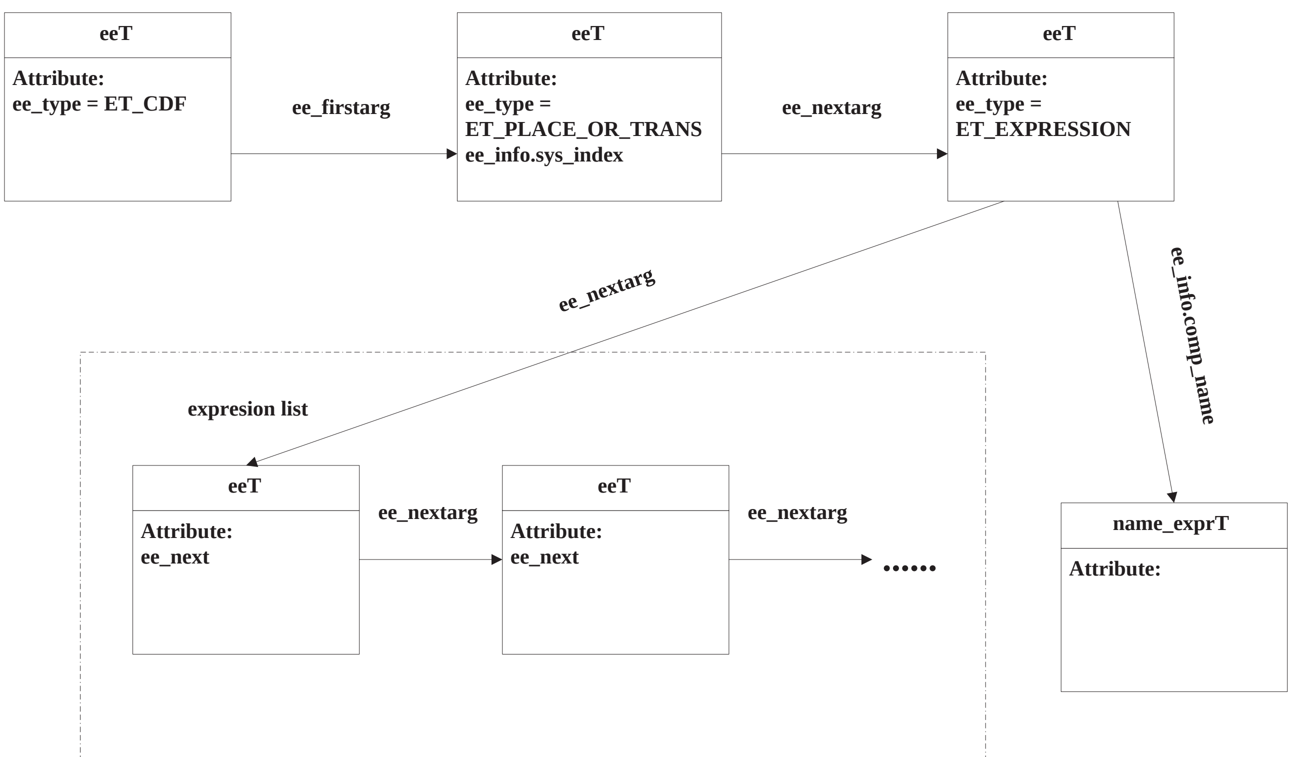


Advanced Expression I

Expression List:
expression, expression, expression, ...
OR
expression
expression
...

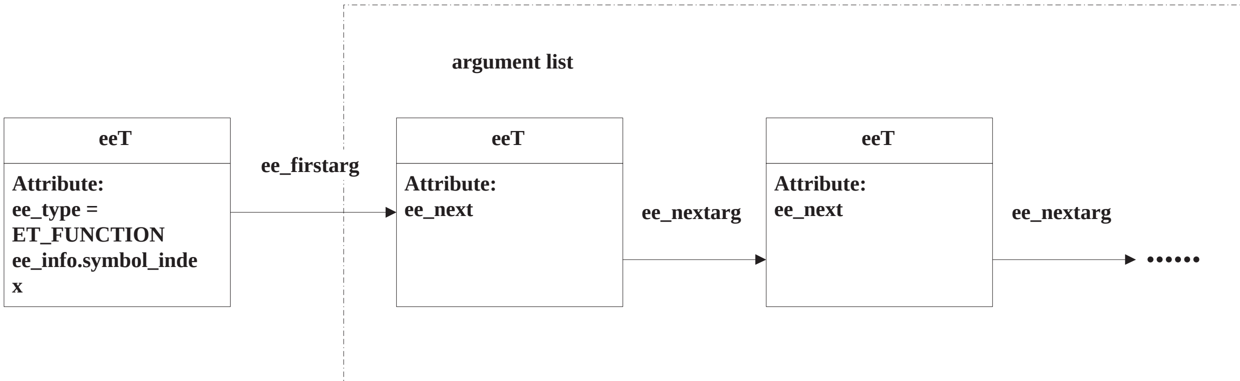


Buildin Function Node:
CDF(gspn_g, node1; a, b c)

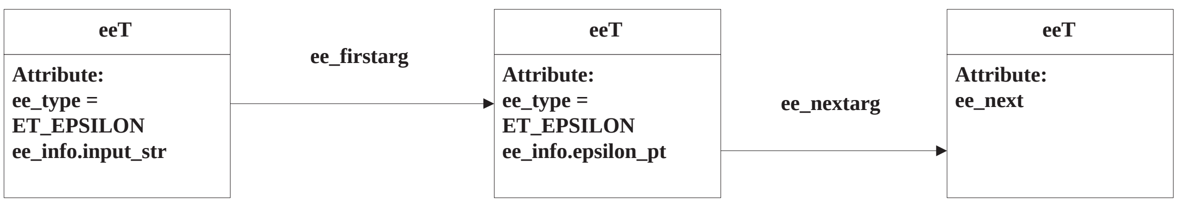


Advanced Expression II

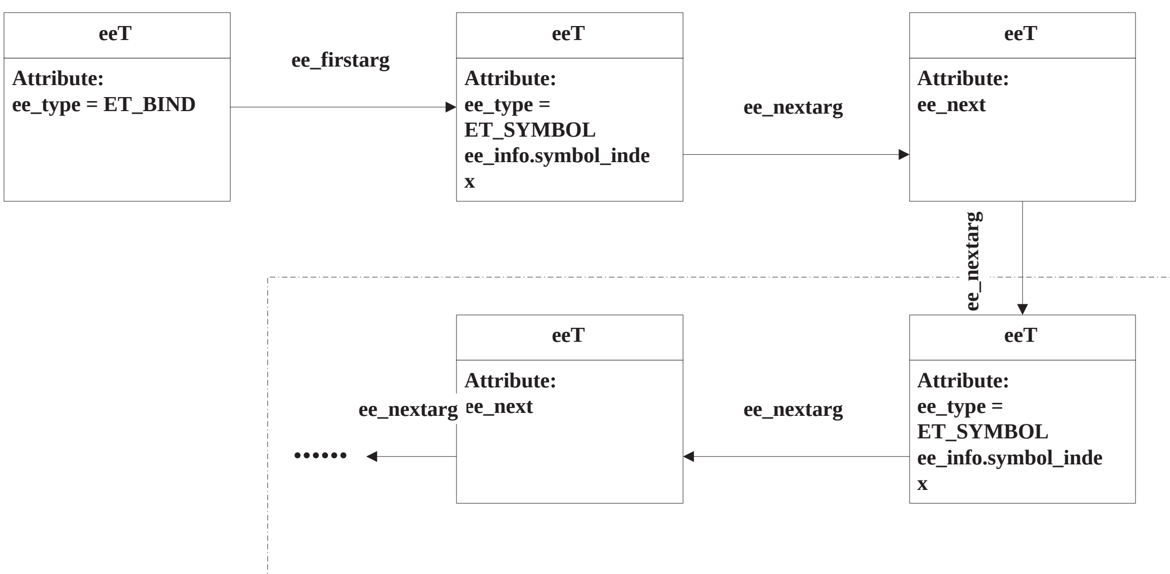
User Defined Function



epsilon *epsilon_id expression*

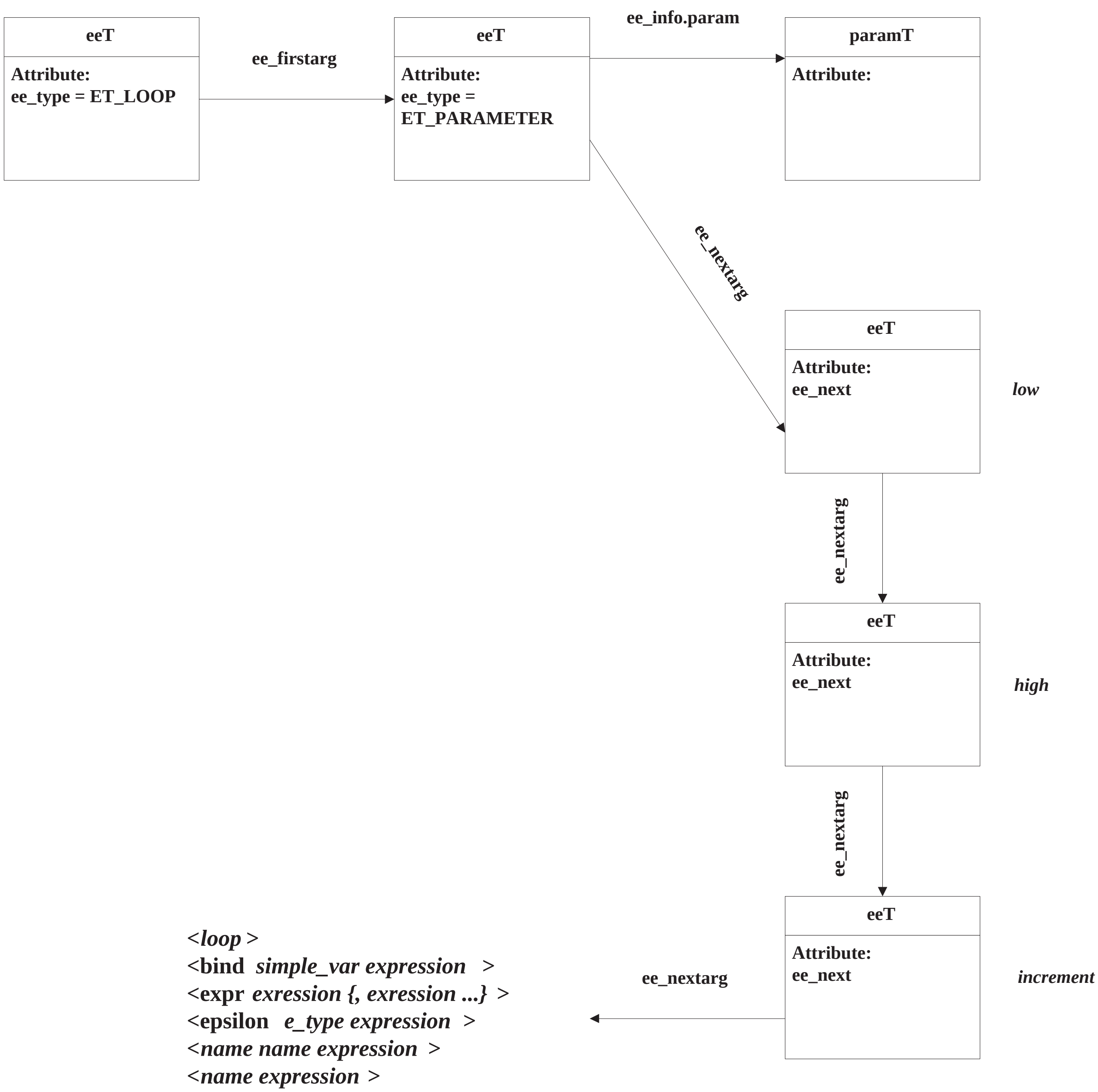


bind *simple_var expression*
OR
bind
<simple_var expression >
end



Advanced Expression III

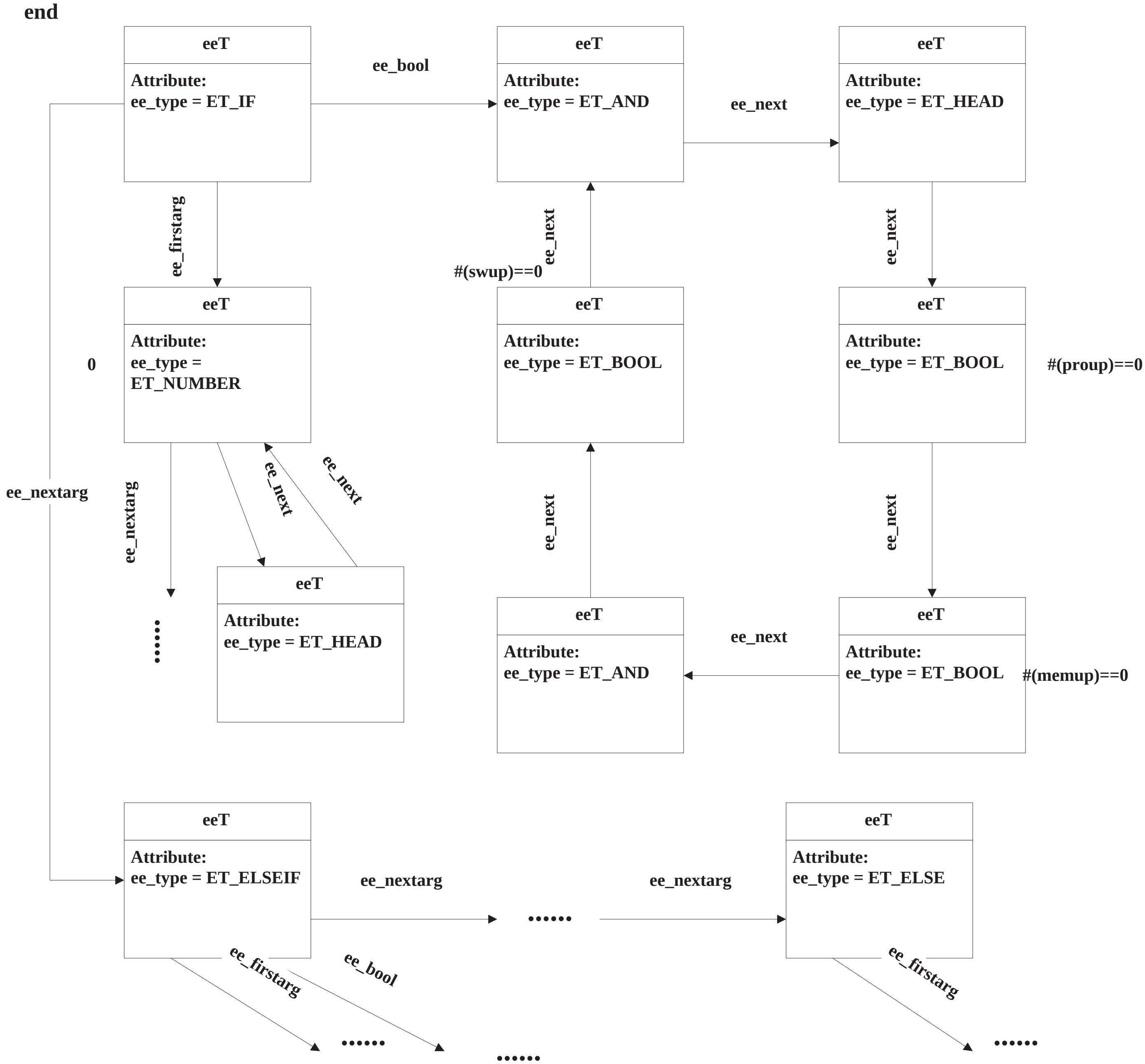
loop ***simple_var, low, high {, increment}***
<<*loop* > | <while_ *statement* > | <bind *simple_var expression* > | <expr *expression {, expression ...}* > | <epsilon *e_type expression* >>
end



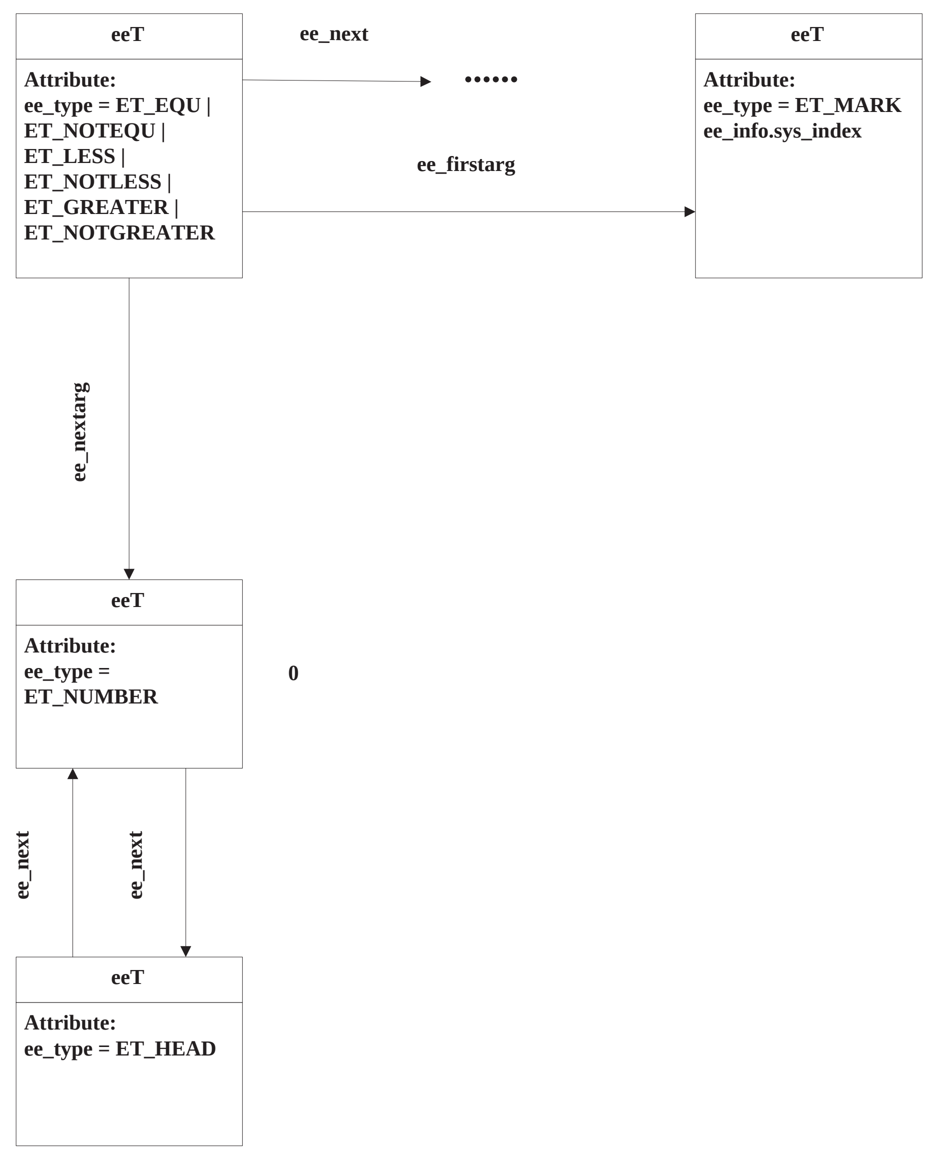
Advanced Expression IV

Extended Expression I ---- if- *statement*

if ((#(procup) == 0) and (#(memup) == 0) and (#(swup) ==0))
0
elseif (......)
<<if- *statement* >|<bind *simple_var expression* > | < *expression* > | <epsilon *e_type expression* >>
else
<<if- *statement* >|<bind *simple_var expression* > | < *expression* > | <epsilon *e_type expression* >>
end



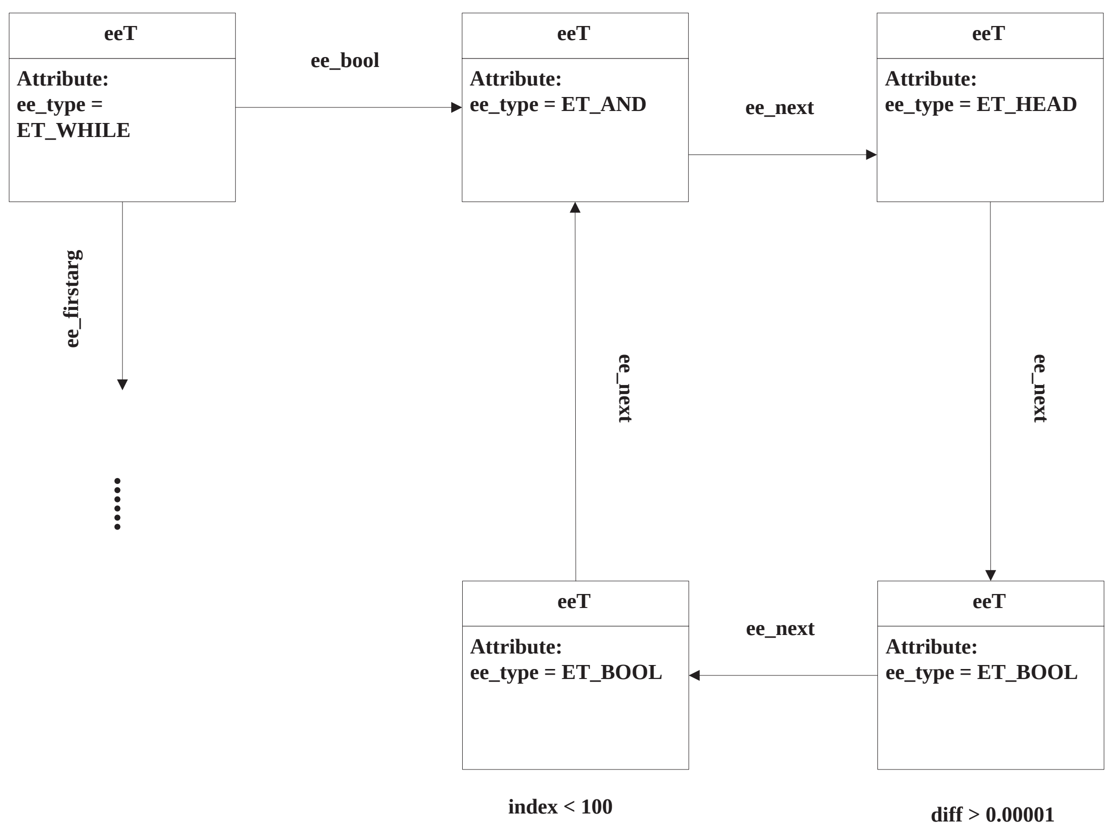
Extended Expression II
#(procup)==0



Advanced Expression VI

Extended Expression II ---- while- *statement*

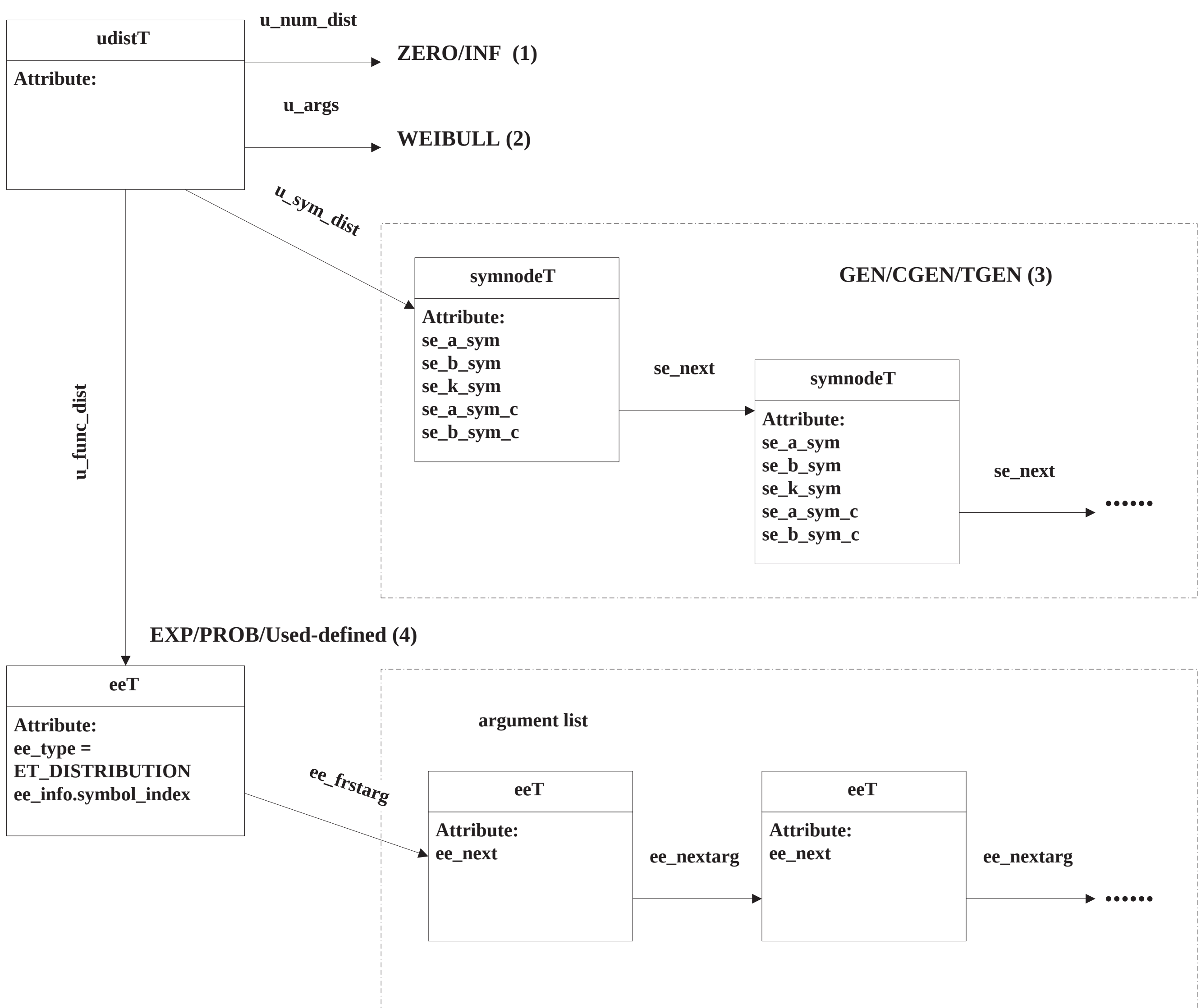
while (diff > 0.00001 and index < 100)
<<while- *statement* >
<*loop* >
<if- *statement* >
<bind *simple_var expression* >
<expr *expression* {, *expression* ... }>
<*expression* >
<epsilon *e_type expression* >>
end



Advanced Expression VII

Distribution Expression:

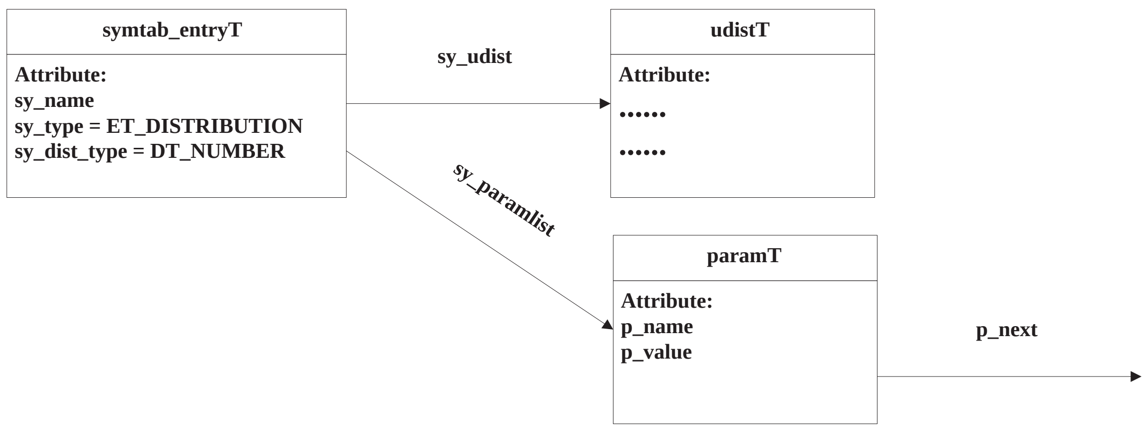
ZERO/INF (1)
WEIBULL (2)
GEN/CGEN/TGEN (3)
EXP/PROB/Used-defined (4)



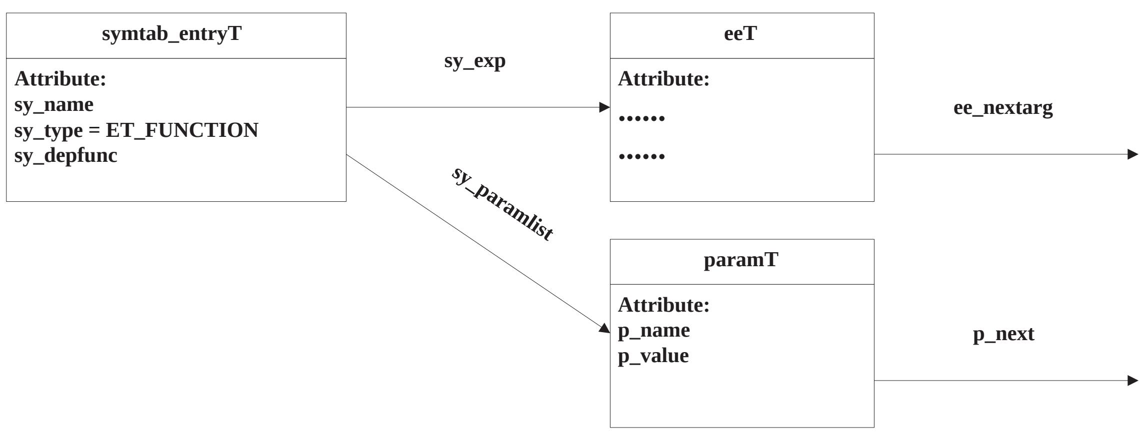
Symbol Table

symP symtab

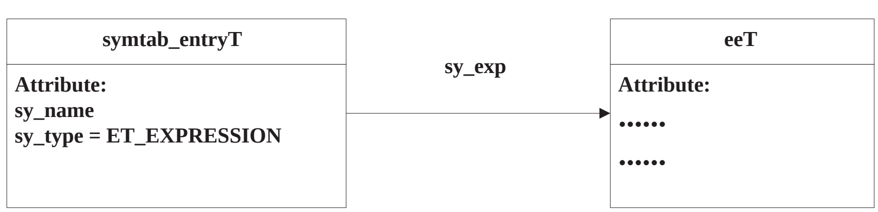
Distribution Function:



User Defined Function:

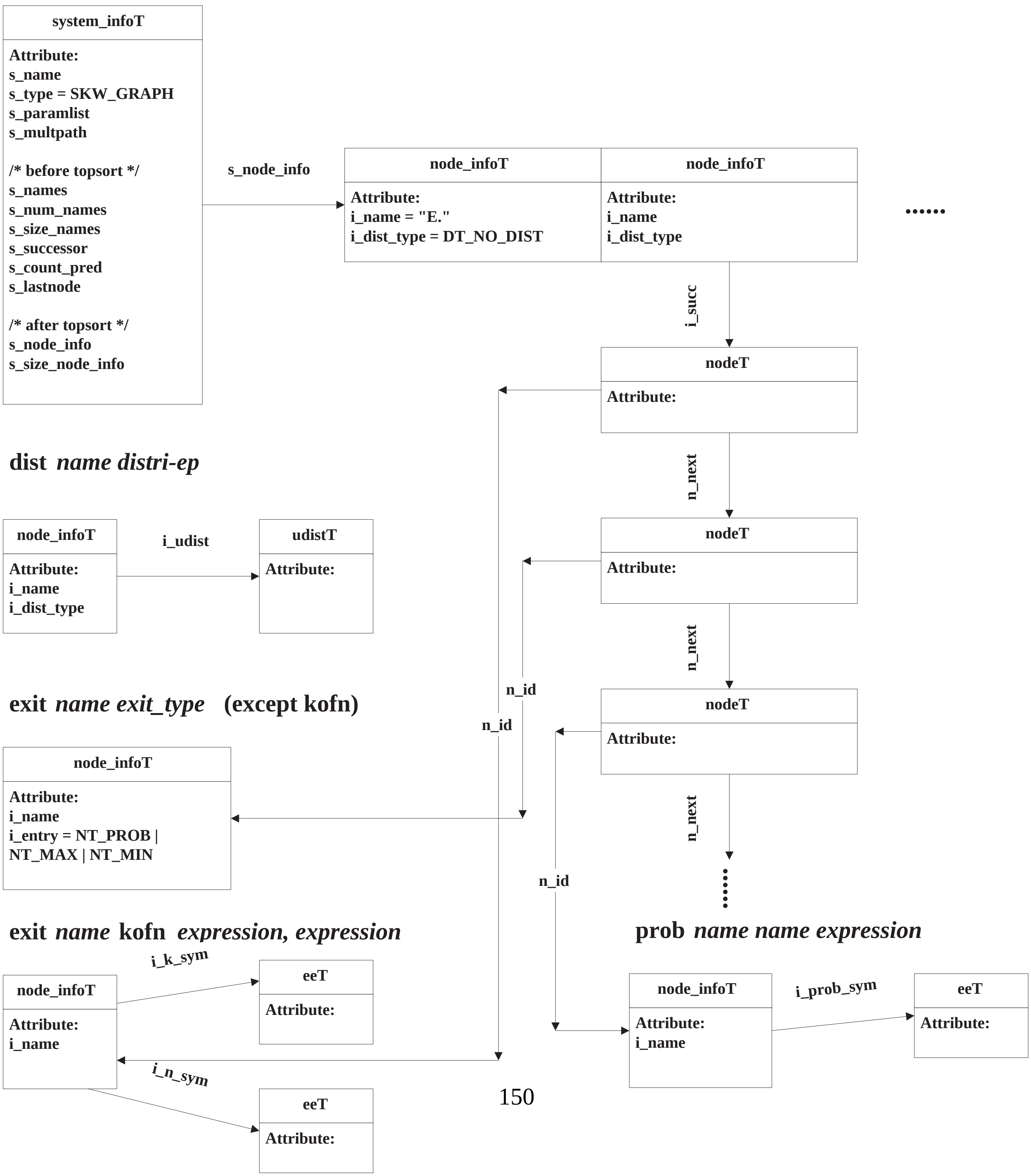


var *defined_var expression*



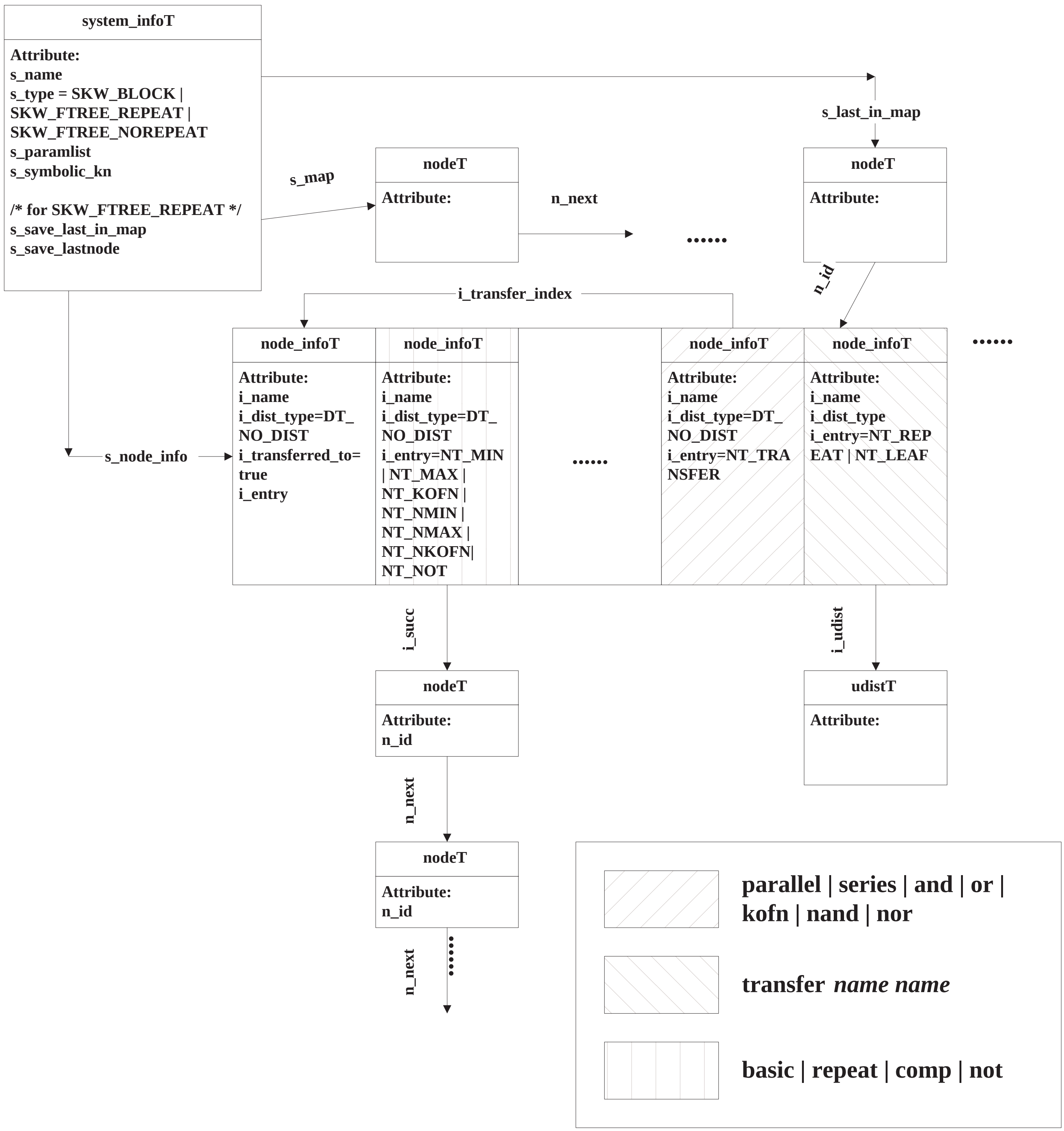
System - Graph

system_infoP system_info

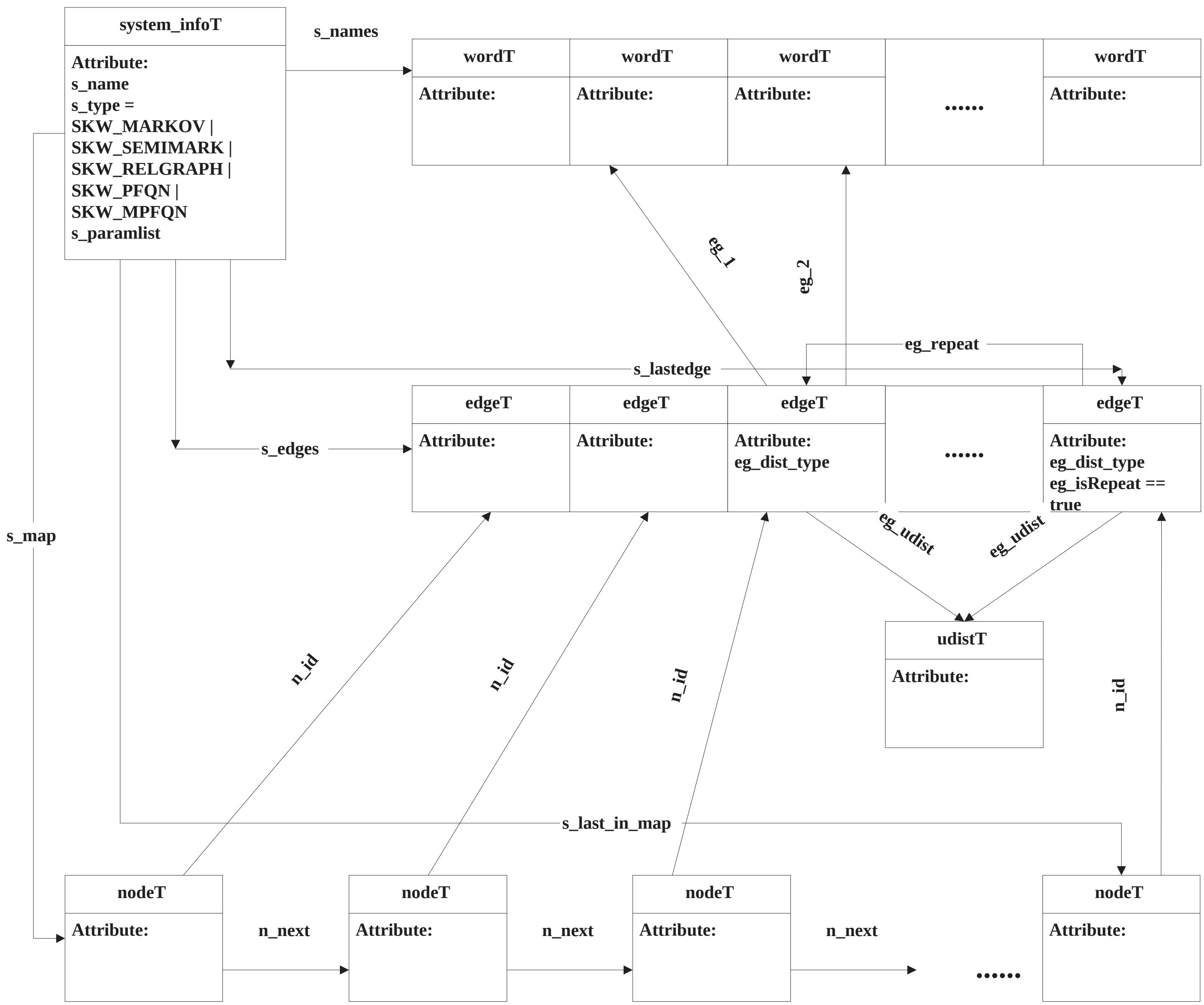


System - Block | Fault Tree | MFT

system_infoP system_info

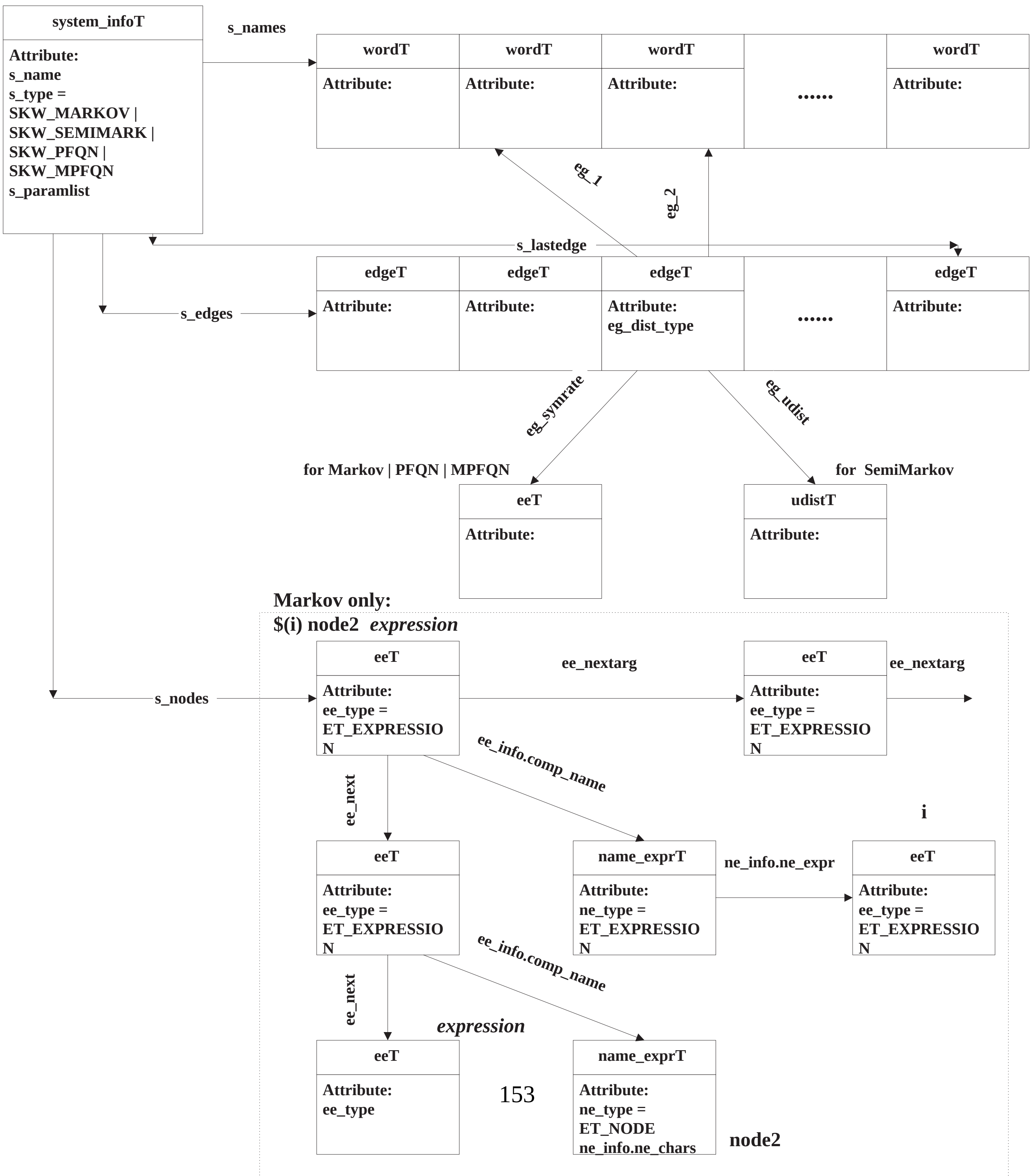


System - Reliability Graphs

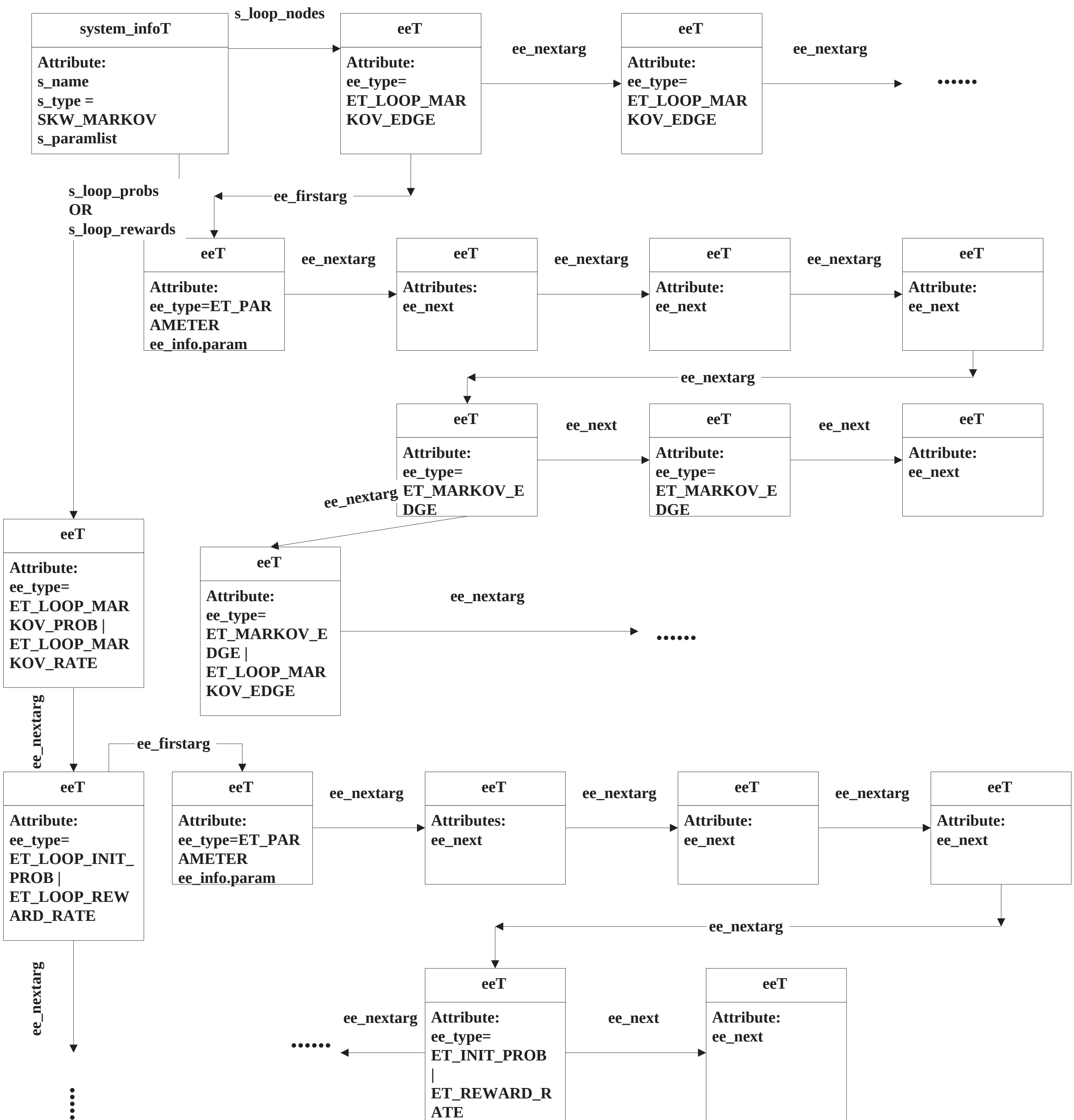


System - Markov Chain*| Semi Markov Chain | PFQN* | MPFQN*

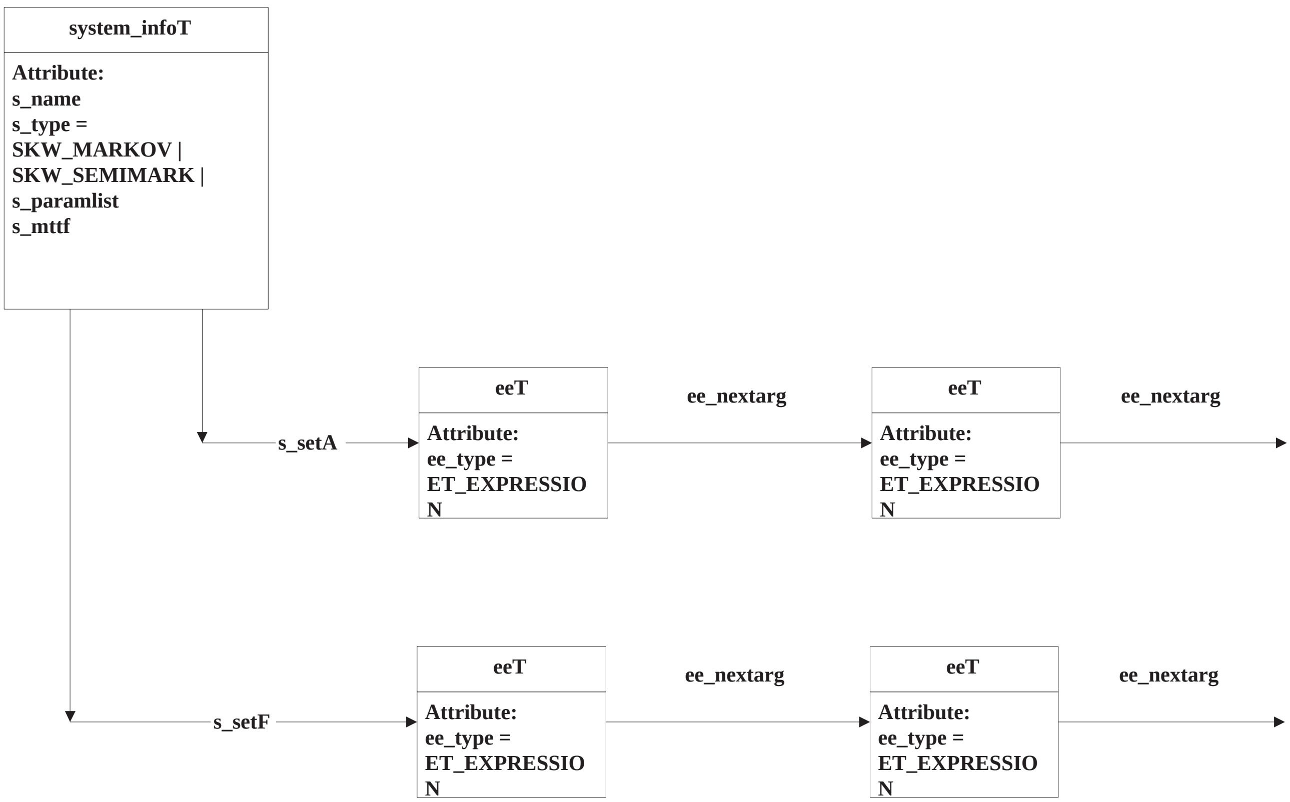
* means basic or partial data structure



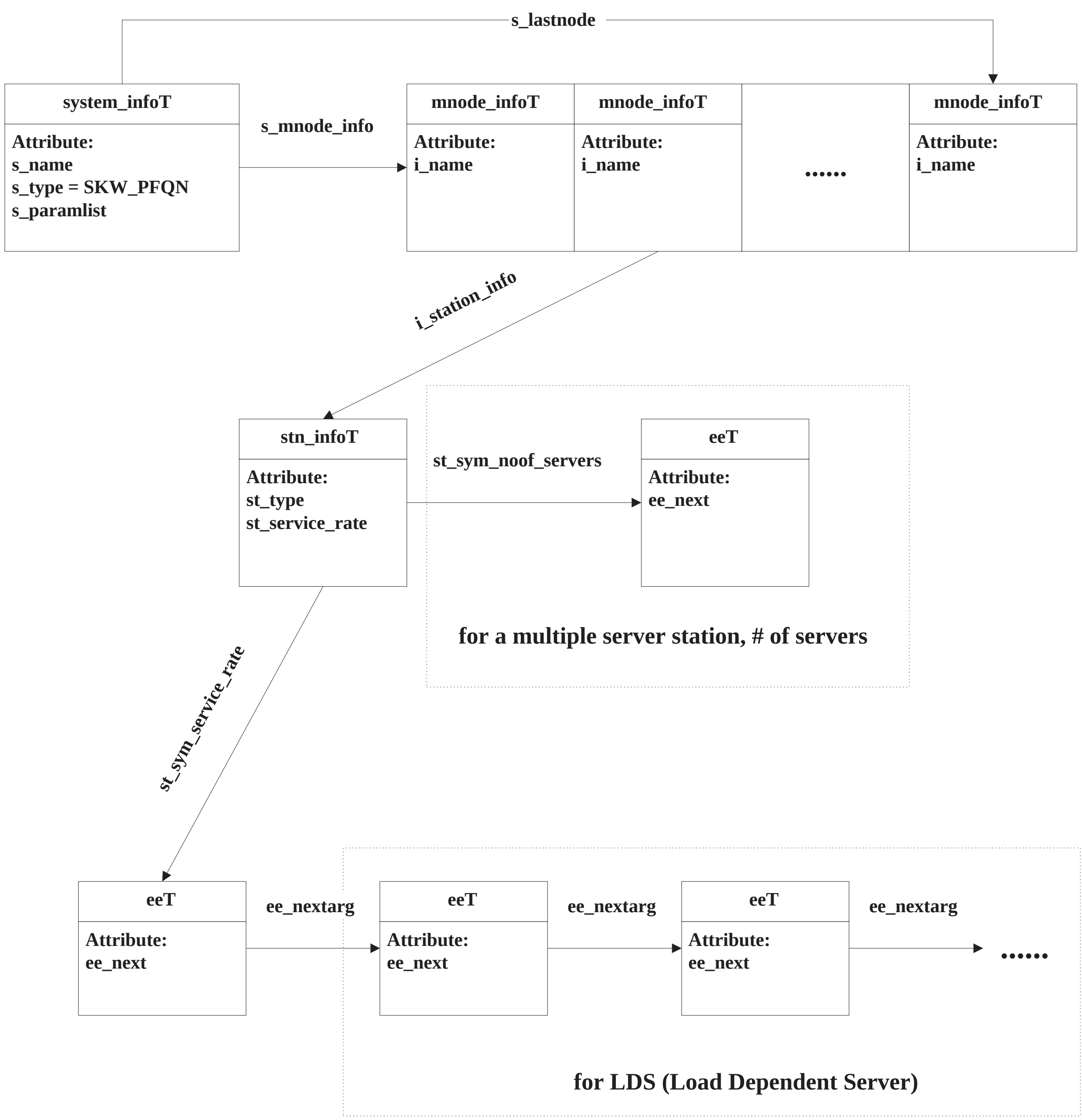
System - Loop in Markov Chain



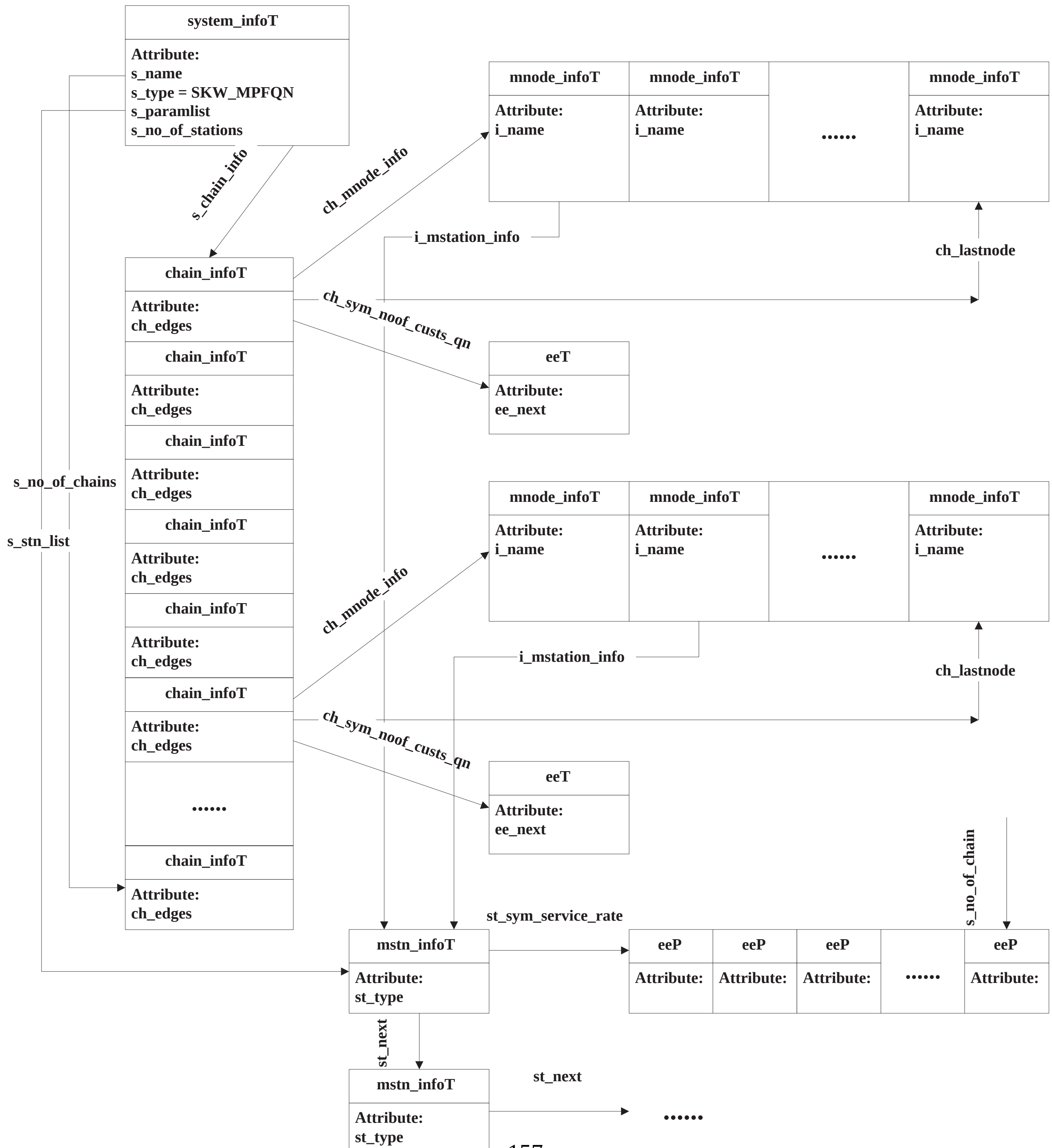
System - Fast MTTF in Markov Chain| Semi-Markov



System - PFQN after mtopsort()

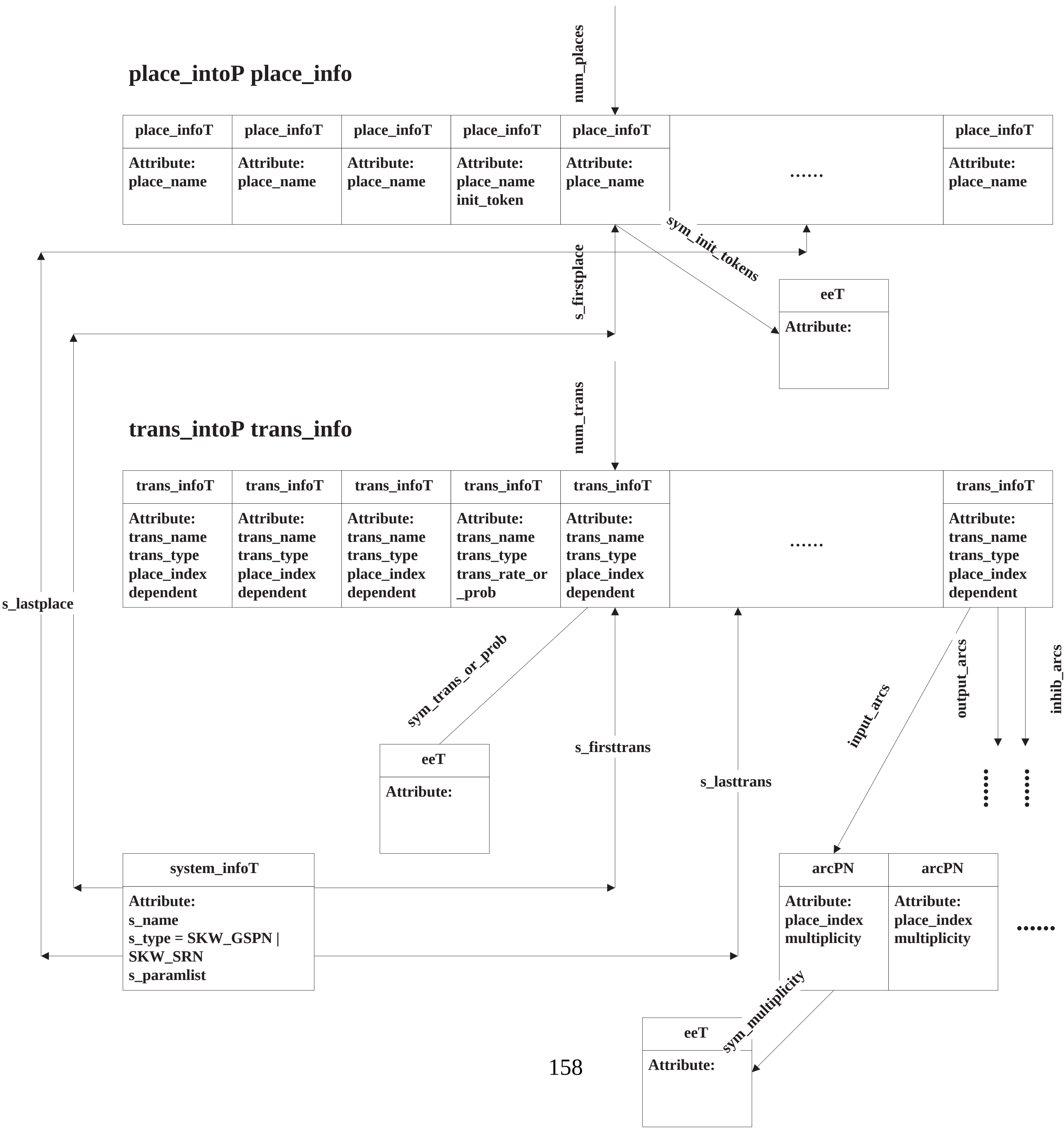


System - MPFQN after mtopsort()

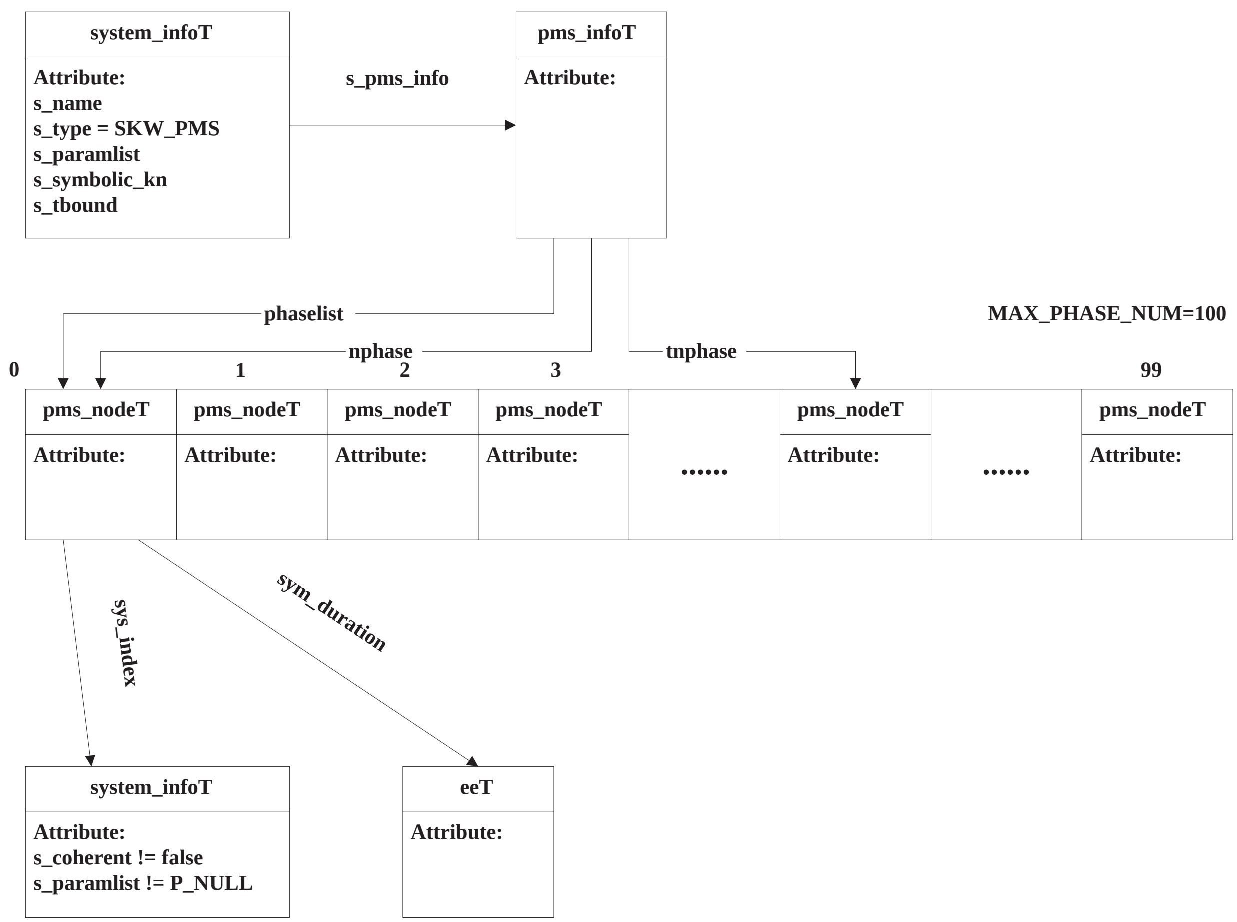


System - GSPN | SRN

system_infoP system_info



System - PMS

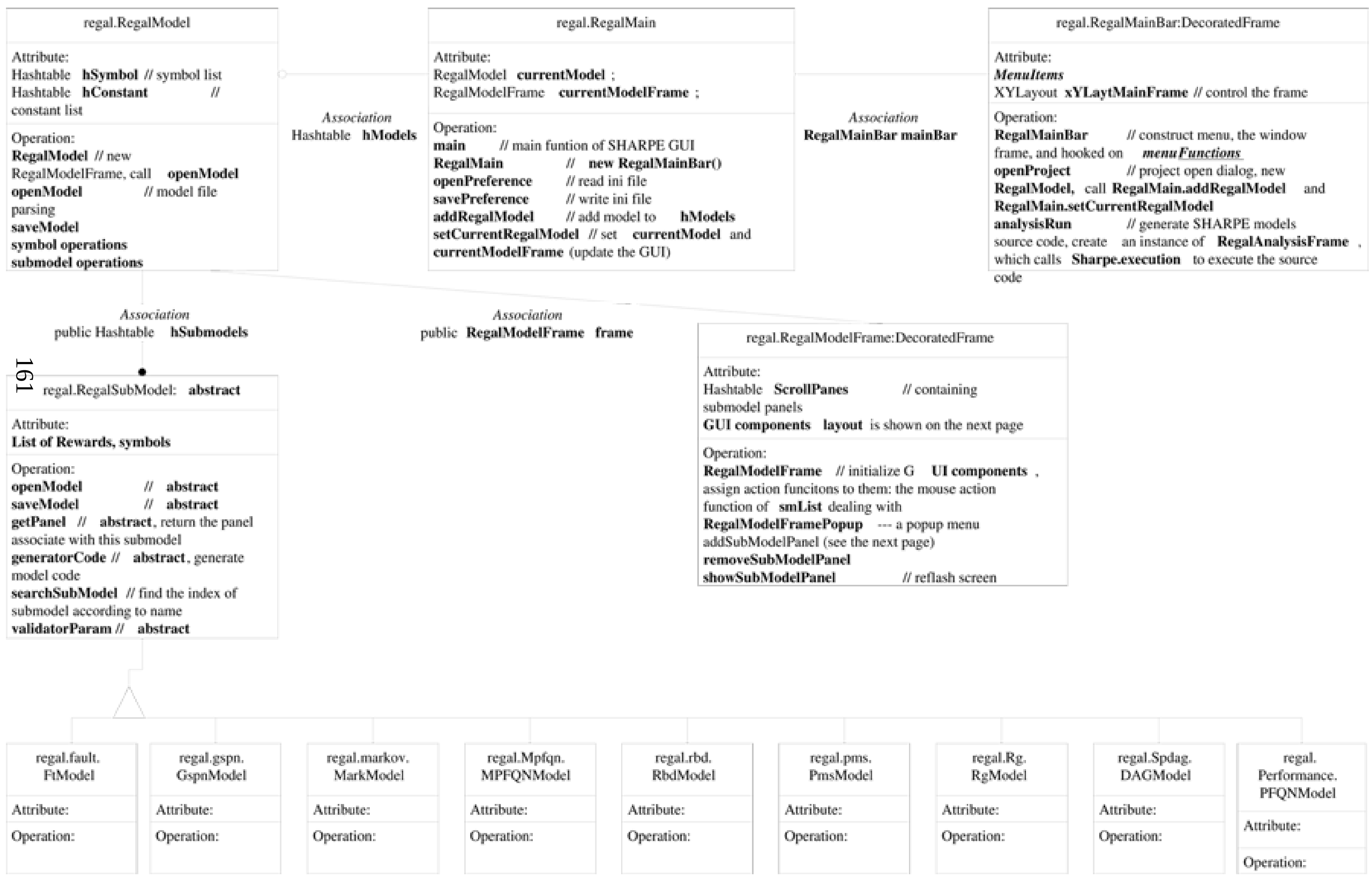

system_infoT
Attribute:
s_name
s_type = SKW_PMS
s_paramlist
s_symbolic_kn
s_tbound
s_pms_info
pms_infoT
Attribute:
phaselist
nphase
tnphase
MAX_PHASE_NUM=100
0
1
2
3
99
pms_nodeT
Attribute:
pms_nodeT
Attribute:
pms_nodeT
Attribute:
pms_nodeT
Attribute:
......
pms_nodeT
Attribute:
......
pms_nodeT
Attribute:
sys_index
sym_duration
system_infoT
Attribute:
s_coherent != false
s_paramlist != P_NULL
eeT
Attribute:

Appendix B

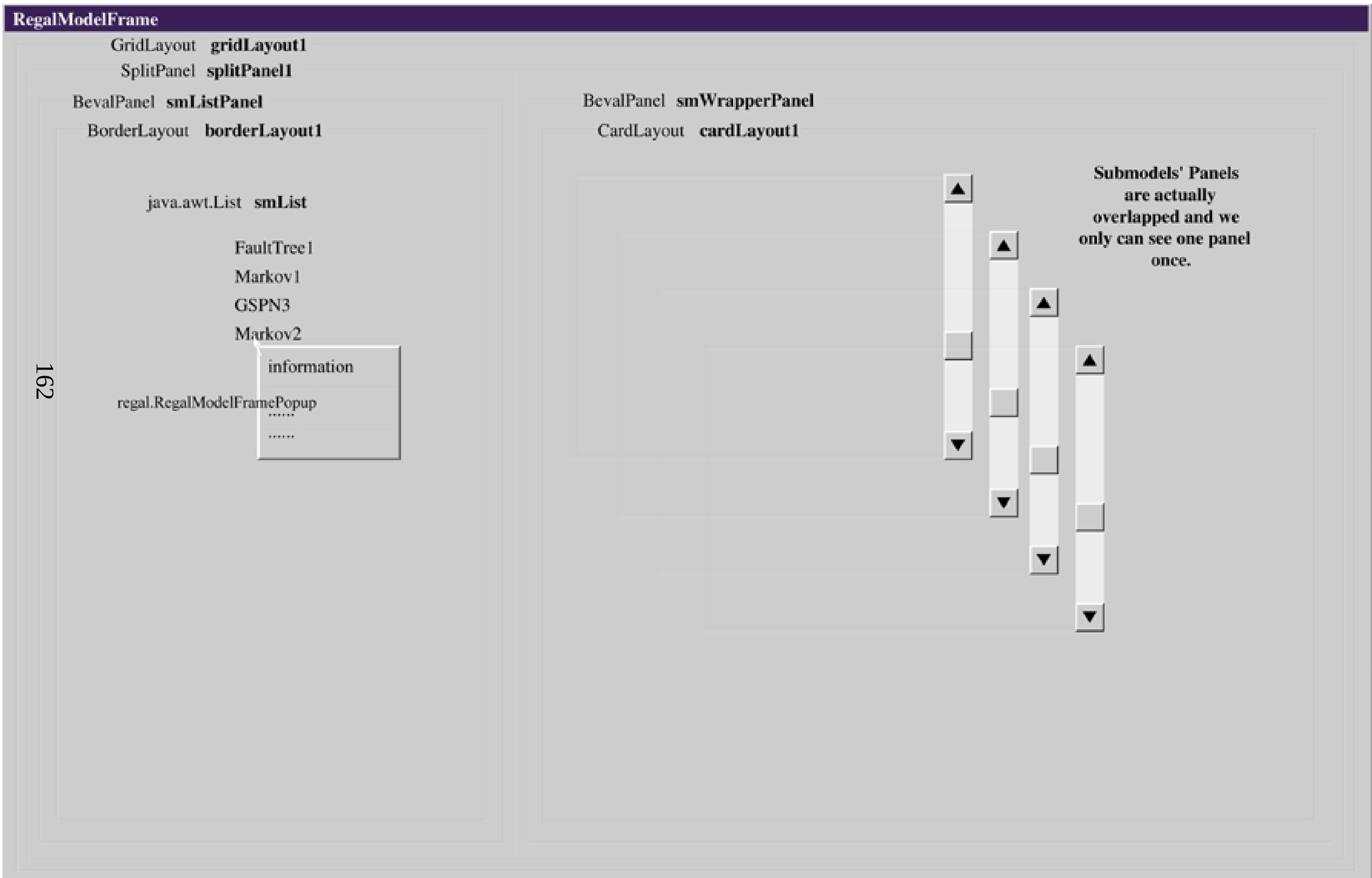
SHARPE GUI Documentation

This appendix is a partial SHARPE GUI document. The first page is the object model [13] of the GUI program. The second is the window layout of the main window. The third is the object model of the analysis window. The last is the window layout of the analysis window.

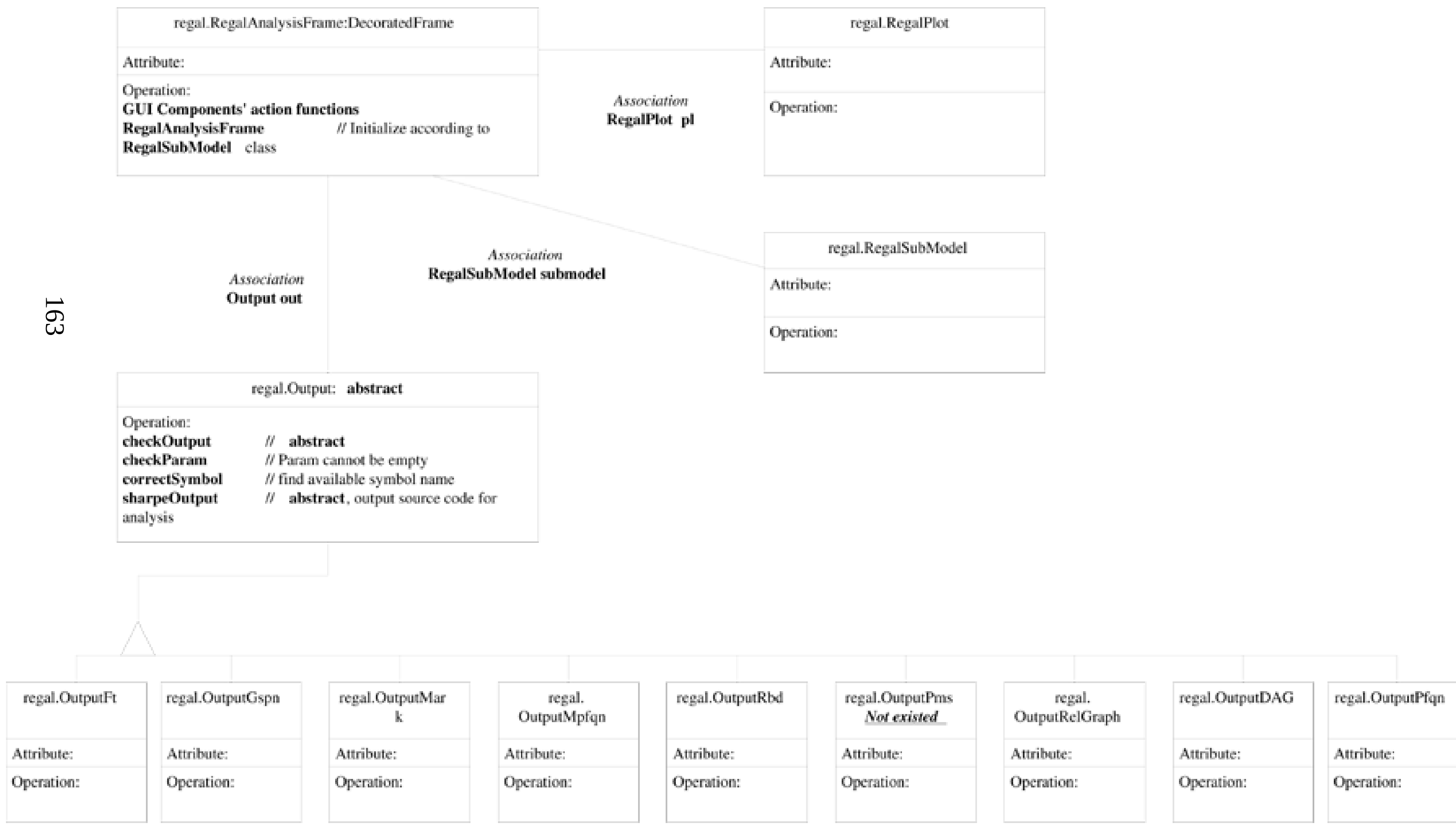
SHARPE GUI Class Architecture I
Basic Architecture



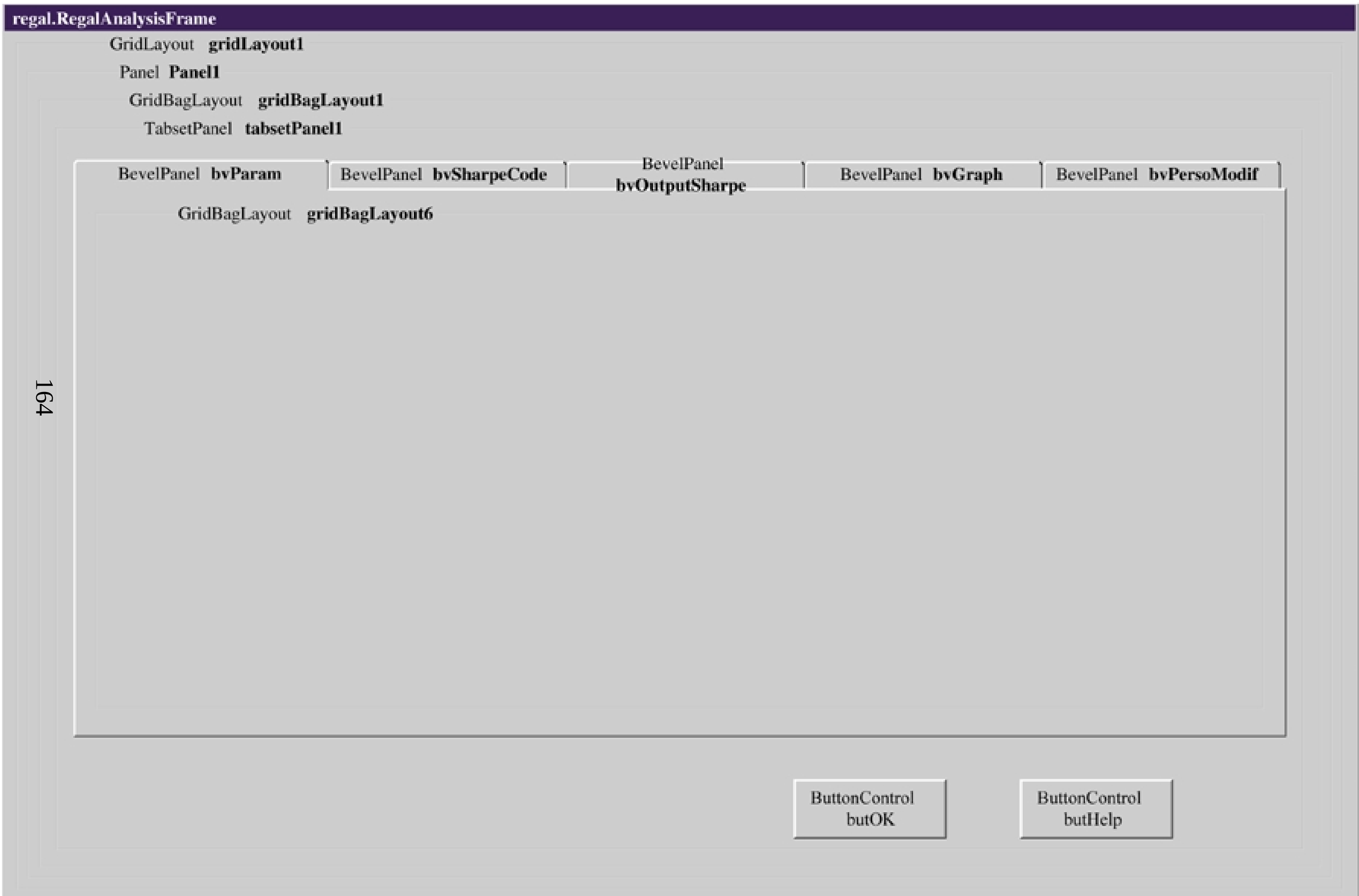
SHARPE GUI Class Architecture II
GUI Components' Layout in regal.RegalModelFrame



SHARPE GUI Class Architecture III
Class regal.RegalAnalysisFrame



SHARPE GUI Class Architecture IV
GUI Components' Layout in regal.RegalAnalysisFrame



Appendix C

SHARPE Examples

Here, more SHARPE examples are listed.

C.1 Fault Tree Examples

C.1.1 SHARPE File — *ftree_n/example12*

```
*Example 12
*Author Luo Tong
*To test the MVI in fault tree with inverse gates
* TEST_KEY sysunrel: 3.0000e-01

* version using only repeated components
ftree ft
repeat a prob(0.3)
repeat b prob(0.4)
basic c prob(0.8)
and d a b
nand f a d
or e d b
or g f e
and h a g
nor i g c
or z h i
end

var sysunrel pzero(ft)
```

```
expr sysunrel

end
```

C.1.2 SHARPE File — *ftree_n/xnkofn1*

```
ftree kn1
repeat r exp(3.2)
basic a exp(7)
basic b exp(4)
basic c exp(5)
basic d exp(11)
kofn abcd 2,4, a b c d
not nabcd abcd
and top nabcd r
end

ftree kn2
repeat r exp(3.2)
basic a exp(7)
basic b exp(4)
basic c exp(5)
basic d exp(11)
nkofn abcd 2,4, a b c d
and top abcd r
end

cdf(kn1)
cdf(kn2)
end
```

C.1.3 SHARPE File — *ftreebdd1/mincut*

```
ftree dsp70
basic a prob(q)
basic b prob(q)
basic c prob(q)
basic d prob(q1)
or t3 a b
and t1 t3 d
transfer d1 d
and t2 c d1
or t0 t1 t2
end

bind
q 0.25
q1 0.30
end

mincuts(dsp70)

expr sysprob(dsp70)

ftree f_long
basic a0123456789012345678901234567890123456789 exp(3.1)
basic b exp(4.2)
basic c exp(3.7)
basic d exp(7.2)
basic e exp(2)
basic f exp(1.2)
basic g exp(0.8)
basic x exp(3)
basic y exp(4)
```

```
transfer e1 e
or BE b e
and A a0123456789012345678901234567890123456789 BE
kofn K1 1,3, a0123456789012345678901234567890123456789 x y
kofn K2 2,4, g c d a0123456789012345678901234567890123456789
or EG e1 g
or FC f c
and E EG FC
or top A K1 K2 E
end

mincuts(f_long)

expr mean(f_long)

ftree f_repeat (k1,k2)
basic a exp(3.1)
basic b exp(4.2)
basic c exp(3.7)
basic d exp(7.2)
basic e exp(2)
basic f exp(1.2)
basic g exp(0.8)
basic x exp(3)
basic y exp(4)
transfer e1 e
or BE b e
and A a BE
kofn K1 k1,3, a x y
kofn K2 k2,4, g c d a
or EG e1 g
or FC f c
```

```
and E EG FC
or top A K1 K2 E
end

mincuts(f_repeat;1,2)

expr mean(f_repeat;1,2)
end
```

C.1.4 SHARPE File — *ftree_bdd2/impt*

```
verbose on

ftree tree0(x)
repeat c1 exp(0.1)
basic c2 exp(0.2)
basic c3 exp(x)
basic c4 exp(0.1)
and and1 c1 c2
and and2 c3 c4
or top and1 and2
end

bdd off
cdf(tree0;0.3)

bdd on
expr bimpt(2; tree0, c1;0.3)
expr bimpt(2; tree0, c1;0.2)
expr bimpt(2; tree0, c1;0.1)
expr bimpt(2; tree0, c1;0.3)
expr cimpt(2; tree0, c1;0.3)
```

```
expr simpt(tree0, c1;0.3)

cdf(tree0;0.3)
end
```

C.2 Examples of Reliability Graphs

C.2.1 SHARPE File — *relgraphbdd2/mincuts*

```
relgraph bridge
1 2 exp(1)
1 3 exp(2)
2 3 exp(3)
3 2 exp(2.3)
2 4 exp(4.7)
3 4 exp(5)
end

mincuts(bridge)

end
```

C.2.2 SHARPE File — *relgraph/minpath*

```
bdd off

relgraph bridge0
1 2 prob(q)
1 3 prob(q)
2 3 prob(q)
```

```
3 2 prob(q)
2 4 prob(q)
3 4 prob(q)
end

bind
q 0.1
end

minpaths(bridge0)

expr 1−sysprob(bridge0)

end
```

C.2.3 SHARPE File — *relgraphbdd2/reltest1*

```
format 8

relgraph bridge0
1 2 prob(q1)
2 4 prob(q2)
1 3 prob(q1)
3 4 prob(q2)
bidirect
2 3 prob(q3)
end

bind
q1 0.01
q2 0.015
q3 0.02
```

```
end

expr sysprob(bridge0)
expr simpt(bridge0, 3, 4)
expr simpt(bridge0, 1, 2)
expr simpt(bridge0, 2, 4)
expr simpt(bridge0, 2, 3)
expr simpt(bridge0, 3, 2)
expr bimpt(10; bridge0, 3, 4)
expr cimpt(10; bridge0, 3, 4)

end
```

C.3 Examples of Fast MTTF [6]

C.3.1 SHARPE File (Markov Chain) — *fastmttf/m6*

```
format 8
bind lambda 0.1
bind mu 1

markov t2 readprobs
6_0 5_1 6*lambda

5_1 5_0 1*lambda
5_1 4_2 5*lambda

5_0 4_1 5*lambda
5_0 6_0 mu

4_2 3_3 4*lambda
4_2 4_1 2*lambda
```

4_1 3_2 4*lambda

4_1 4_0 1*lambda

4_1 5_1 mu

4_0 3_1 4*lambda

4_0 5_0 mu

3_3 2_4 3*lambda

3_3 3_2 3*lambda

3_2 2_3 3*lambda

3_2 3_1 2*lambda

3_2 4_2 mu

3_1 2_2 3*lambda

3_1 3_0 1*lambda

3_1 4_1 mu

3_0 2_1 3*lambda

3_0 4_0 mu

2_4 1_5 2*lambda

2_4 2_3 4*lambda

2_3 1_4 2*lambda

2_3 2_2 3*lambda

2_3 3_3 mu

2_2 1_3 2*lambda

2_2 2_1 2*lambda

2_2 3_2 mu

2_1 1_2 2*lambda

2_1 2_0 1*lambda

2_1 3_1 mu

2_0 1_1 2*lambda

2_0 3_0 mu

1_5 0_6 1*lambda

1_5 1_4 5*lambda

1_4 0_5 1*lambda

1_4 1_3 4*lambda

1_4 2_4 mu

1_3 0_4 1*lambda

1_3 1_2 3*lambda

1_3 2_3 mu

1_2 0_3 1*lambda

1_2 1_1 2*lambda

1_2 2_2 mu

1_1 0_2 1*lambda

1_1 1_0 1*lambda

1_1 2_1 mu

1_0 0_1 1*lambda

1_0 2_0 mu

0_6 0_5 6*lambda

0_5 0_4 5*lambda

0_5 1_5 mu

```
0_4 0_3 4*lambda
0_4 1_4 mu

0_3 0_2 3*lambda
0_3 1_3 mu

0_2 0_1 2*lambda
0_2 1_2 mu

0_1 1_1 mu
0_1 0_0 1*lambda

0_0 1_0 mu

end
6_0 1
end
fastmttf
6_0 READA
2_4 READA
*3_3 READA
0_0 READF
end


expr fastmttf(t2)
end
```

C.3.2 SHARPE File (Semi-Markov Chain) — *fastmttf/semit*

```
semimark abc2
```

```
m1 m2 exp(1.2)
m2 m3 exp(0.8)
m1 m3 exp(1.4)
m2 m1 exp(0.3)
m3 m1 exp(1.5)
m3 m4 exp(2.5)
m4 m1 exp(1.0)
end
m1 1
end
fastmttf
m1 READA
m2 READA
m3 READF
end


expr fastmttf(abc2)
end
```

C.4 SRN Example

C.4.1 SHARPE File — *srn/mtta*

```
* Translate from sensi.c of SPNP6

format 8

bind
thinktime   1
CPUrate     0.01
```

```
rate1       0.04
rate2       0.05
TK       2
exit_prob    0.01
out1_prob    0.30
out2_prob    0.69
lambda       1.0/CPUrate
theta       1.0
end

srn mttatest()
* Places
think    0
CPU    TK
decide    0
use1    0
use2    0
end
* Timed transitions
go    placedep    think    1.0/thinktime
CPUdone    ind    lambda
done1    ind    1.0/rate1*theta
done2    ind    1.0/rate2*theta
end
* Immediate transitions
exit1    ind    exit_prob
out1    ind    out1_prob
out2    ind    out2_prob
end
* Input arcs
think    go    1
CPU    CPUdone    1
```

```
decide   exit1   1
decide   out1   1
decide   out2   1
use1   done1   1
use2   done2   1
end
* Output arcs
go   CPU   1
CPUdone   decide   1
exit1   think   1
out1   use1   1
out2   use2   1
done1   CPU   1
done2   CPU   1
end
* Inhibitor arcs
think   go   TK
end

func   refunc()
if (#(think) == TK)
0
else
1
end
end

expr  srn_cexrinf(mttatest; refunc)
expr   mtta(mttatest)

end
```

Bibliography

[1] B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[2] H. Choi and K. S. Trivedi. Approximate performance models of polling systems using stochastic Petri nets. In *Proceedings of IEEE Infocom 92, 11th Annual Joint Conference of the IEEE Computer and Communication Societies*, Florence Italy, May 1992.

[3] G. Ciardo, J. Muppala, and K. S. Trivedi. Analyzing concurrent and fault-tolerant software using stochastic reward nets. *Journal of Parallel and Distributed Computing*, 15:255–269, 1992.

[4] W. Fischer and K. Meier-Hellstern. The markov-modulated poisson process (mmpp) cookbook. *Performance Evaluation*, 18(2):149–171, 1992.

[5] G. Haring, R. Marie, R. Puigjaner, and K. S. Trivedi. Loss formulae and their optimization for cellular networks. *IEEE Transactions on Vehicular Technology (to appear)*, 1999.

[6] P. Heidelberger, J. K. Muppala, and K. S. Trivedi. Accelerating mean time to failure computations. *Performance Evaluation*, 27 28:627–645, Oct. 1996.

[7] D. Hong and S. S. Rappaport. Traffic model and performance analysis for cellular mobile radio telephone systems with prioritized and nonprioritized handoff procedures. *IEEE Trans. Veh. Technol.*, 35(3):77–99, Aug. 1986.

[8] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[9] Y. B. Lin, S. Mohan, and A. Noerpel. Queueing priority channel assignment strategies for pcs hand-off and initial access. *IEEE Trans. on Vehi. Tech.*, 43(3):704–712, Aug.

[10] M. Ajmone Marsan, G. Balbo, and G. Conte. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transitions on Computer Systems*, 2:93–122, May 1984.

[11] I. Mura, A. Bondavalli, X. Zang, and K. S. Trivedi. Dependability modelling and evaluation of phased mission systems: a DSPN approach. In *Proc. IFIP International Conference on Dependable Computing for Critical Applications (DCCA-7)*, pages 299–318. San Jose, California, Jan. 1999.

[12] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Bonn, Germany, Jan. 1962.

[13] Dr. James Rumbaugh. *OMT Insights*. SIGS Books, 1996.

[14] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using SHARPE Software Package*. Kluwer Academic Publishers, 1995.

[15] W. Stewart. *Introduction to Numerical Solution of Markov Chains*. Princeton University Press, Princeton, N.J., 1994.

[16] K. S. Trivedi. *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Prentice-Hall, Inc., Englewood Cliffs, 1982.

[17] W. Xie. Markov regenerative process in sharpe. Master's thesis, Duke University, Durham, U.S.A., 1999.

[18] X. Zang. *Dependability Modeling of Computer Systems and Networks*. PhD thesis, Duke University, Durham, U.S.A., 1999.