SHARPE: SYMBOLIC HIERARCHICAL AUTOMATED RELIABILITY AND PERFORMANCE EVALUATOR

Introduction and Guide for Users

Robin A. Sahner
308 W. Delaware
Urbana, IL 61801, USA

Kishor S. Trivedi
1713 Tisdale St.
Durham, NC 27705, USA

February 1992

Chapter 1

Introduction

SHARPE (Symbolic Hierarchical Automated Reliability and Performance Evaluator) is a software tool that analyzes stochastic models. It was developed for three groups of users: practicing engineers [20, 23], researchers in performance and reliability modeling [26] and students in engineering/science courses [46].

When a computer system design is even moderately complex, it is hard to predict the performance and reliability characteristics of the resulting system. Usually, it is too expensive and time-consuming to build even one system to take measurements. Even when that is not the case, it has been argued [21] that measurements guided by a system model result in a more effective design methodology. If the model is a good abstraction of the system, the analyst can quickly carry out trade-off studies, answer "what if" questions, perform sensitivity analysis and compare design alternatives. A tool for easily specifying and quickly solving stochastic models can be a useful aid in the repertoire of a system designer [20].

Researchers in computer architecture and fault-tolerant computing often wish to evaluate their design ideas before committing to a prototype or an expensive simulation. Likewise, researchers interested in various approximations involving decomposition, state truncation, etc., need a tool to carry out experiments. SHARPE has been used by various researchers for this purpose [23, 26, 34, 51].

Computer software can be a very useful aid in the teaching of mathematics, computer science and engineering. This includes involving students in both writing and using software that illustrates the subject matter. As examples, we cite the extensive use of Mathematica [1] in all kinds of diverse courses and an increasing use of software in the teaching of calculus [49, 42, 17]. SHARPE is being used by many universities for the teaching of fault-tolerant computing, performance evaluation, reliability engineering and applied probability.

The most common type of stochastic model used in practice is discrete-event simulation, because fairly detailed system behavior can be captured in such models. But since running simulations can sometimes be very time-consuming, it is useful for system designers to use analytic models instead of or in addition to simulation models. Due to recent developments in model generation, model hierarchies and solution techniques and their implementation in tools such as SHARPE, large and realistic models can be developed and studied.

Dependability, performance, and performability modeling techniques provide a useful method for understanding the dynamic behavior of a computer or communication system. To be useful, the models should reflect important system characteristics such as fault-tolerance, automatic reconfiguration, and repair; contention for resources; concurrency and synchronization; deadlines imposed on the tasks; and graceful degradation. Furthermore, complexity of current-day systems and corresponding system models should be explicitly addressed.

Reliability, availability, safety, and related measures are collectively known as dependability. Dependability evaluation encompasses fault-tolerance, reconfiguration, and repair aspects of system behavior. Reliability block diagrams, fault trees, and reliability graphs are commonly used to study the dependability of systems [47]. Although these models are concise and have efficient solution methods, they cannot represent dependencies among components [46] as easily as Markov models can [12, 18].

Traditional performance evaluation is concerned with contention for system resources. Performance evaluation of parallel and distributed systems also address concurrency and synchronization of tasks. Real-time system performance evaluation takes into account various hard and soft deadlines on task exection times. Task precedence graphs [29, 45] can be used to model the performance of concurrent programs with unlimited resources.

[1] *Mathematica* is a trademark of Wolfram Reserch Inc.

Product form queueing networks [32, 33, 53], on the other hand, can represent contention for resources. However they cannot model concurrency within a job, synchronization, or server failures, since these violate the product form assumptions. Markov models, on the other hand, allow all these characteristics to be modeled.

Thus, performance analysis assumes a fault-free system while reliability and availability analysis is carried out to study system behavior in the presence of component faults, disregarding different performance levels in different configurations. Several different types of interactions and corresponding tradeoffs have prompted researchers to consider combined evaluation of performance and reliability/availability [36, 55]. Most work on the combined evaluation is based on the extension of Markov chains to Markov reward models [22], where a reward is attached to each state of the Markov process.

Markov reward models have the potential to reflect concurrency, contention, fault-tolerance, and degradable performance; they can be used to obtain not only program/system performance and system reliability/availability measures, but also combined measures of performance and reliability/availability [5, 10, 36, 55].

Models like fault trees, reliability block diagrams, and reliability graphs, used for dependability analysis, and product-form queueing networks and series-parallel graphs, used for performance analysis, are "non-state-space" models. That is, model construction and analysis do not require generation of a "state space", an enumeration of the possibilities of what can happen to the system. However, certain kinds of real-life system structures and dependencies violate the assumptions made by these models. State-space methods (such as those based on Markov and semi-Markov models) can capture system dependencies. They are very widely applicable, but the possibly large size of the state space can be a problem both during model generation and model analysis.

Generalized Stochastic Petri nets (GSPNs) can be used to generate a (large) underlying Markov model automatically starting from a concise description. Nevertheless, the combinatorial growth of their state space (reachability graph) constitutes a major limitation to applicability of GSPN models in real-life problems. A SHARPE user can exploit model hierarchies to compose an overall model solution from individual model results, thus avoiding a large overall state space. We emphasize that hierachy is used in SHARPE not just for model specification but for model solution as well.

Each kind of analytic model is well suited for some particular set of applications, but every model type also has limitations. Because of this tradeoff between ease of solution and ability to capture system details, the mathematical modeler needs a "toolchest" of various model types that can be chosen from freely and combined with each other, as each particular problem dictates. SHARPE is such a toolchest.

SHARPE allows its users to construct and analyze performance, reliability, availability, and performability models. Users can set up and solve a variety of model types, compare results for different models of the same system, see how altering system parameters affects measures of effectiveness of the system, and experiment with modeling techniques, including the use of exact and approximate system or model decomposition. SHARPE can also be used to illustrate problems of large state spaces and stiff systems and to provide examples of methods for avoiding these problems.

In this guide, we will use a series of examples to show how such a toolchest could be used to investigate the characteristics and design tradeoffs. We will also show how we can set up multiple models for the same underlying system. This is a good validity check for both the construction and analysis of the models.

1.1 Overview of SHARPE

SHARPE provides a specification language and analysis algorithms for the following model types:

- reliability block diagrams
- fault trees
- reliability graphs
- series-parallel acyclic directed graphs
- product-form queueing networks
- Markov and semi-Markov chains
- generalized stochastic Petri nets

SHARPE is a modeler's toolchest; it provides a specification language for building single or hierarchical combinations of models and for choosing algorithms for analyzing the models. The models are not, a priori, assumed to be an abstraction of any particular real-world system. To SHARPE, a Markov chain is just a Markov chain, not a model of (for example) a fault-tolerant system where some faults can be repaired and some cannot. The advantage of this is that a user of SHARPE is free to use any of the supported model types in any valid combination. However, this means that it is up to the user to choose models that are an accurate abstraction of the problem under investigation and to interpret the results in a meaningful way for that particular problem.

SHARPE allows the results of the analysis of each model type it provides to be used in the parameterization of the other model types, subject only to validity checks on the parameters. Information carried between models can be either numbers (such as steady-state probabilities, average numbers of tokens in a Petri net's place, or the probability of failure by a time t) or symbolic functions in time t (such as the cumulative distribution function for the time to failure of a system).

The SHARPE models are allowed to be hierarchical in the sense that output from one submodel can be used as input to another. That is, the parameters for a model component can be expressions involving values or functions obtained by the analysis of another model. There are no restrictions on the ways in which models can be combined. Users can choose and and mix model types freely.

SHARPE is written in C. It was developed on UNIX[2] systems but can be compiled on any system that provides a C compiler and the usual C math and input/output libraries. The program has two modes of operation: it can read input from one or more files (batch, or non-interactive mode) or read interactively from a terminal (interactive mode).

1.2 New Features

The previous version of this guide was dated September 1986. Since that time, the following new features have been added to SHARPE:

- the ability to include text from secondary files (**include**)
- the ability to echo an input line to the output (**echo**)
- analysis of irreducible, phase-type and acyclic Generalized Stochastic Petri Nets (**gspn**)
- analysis of single-chain and multiple-chain Product Form Queueing Networks (**pfqn**)
- analysis of reliability graphs
- analysis of fault trees with repeated (shared) components (events)
- analysis of Markov and semi-Markov reward models (**reward**)
- a new built-in function for summation (**sum**)
- a loop mechanism for requesting results (**loop**)
- a new one-line form of the **bind** statement, for use within loops but valid everywhere
- numerical algorithms for finding transient probabilities for Markov chains, fault trees, reliability block diagrams, and reliability graphs (**tvalue**)
- more algorithm choices for phase-type and irreducible Markov chains and GSPNs
- a command to allow the user to set the value of the various error tolerance ("epsilons") used in the algorithms (**epsilon**)

[2] UNIX is a registered trademark of AT&T.

1.3 Contents of this Guide

In Chapter 2, we give a brief description of each of the model types. In Chapter 3, we discuss what kind of distribution functions SHARPE uses and how they are specified. In the following chapters, we present the SHARPE non-interactive input language through a series of examples. The examples are grouped by application areas: reliability and availability analysis (Chapter 4), performance analysis (Chapter 5), hierarchical models (Chapter 6) and performability analysis (Chapter 7).

In Chapter 8, we discuss algorithmic and numerical considerations. In Chapter 9, we describe the SHARPE command line syntax. In Chapter 10, we explain how to use SHARPE interactively. Appendix A gives a complete description of the non-interactive input language. Appendix B provides information about the class of distribution functions used by SHARPE.

Readers interested in background material on probability theory and stochastic modeling may refer to [53]. For background on the basics of GSPN models see [1].

1.4 Some Conventions

In the examples we provide, we will often show input and output files with line numbers at the beginning of each line. Actual SHARPE input and output files do not (and can not) contain line numbers; we have included the numbers in the figures so that we can refer to particular lines more easily.

In the text of this guide (but not in actual SHARPE input files), keywords will appear in boldface. SHARPE is lenient about the placement of words and separators within each line but is very strict about requiring that "statements" appear one per line. This line-at-a-time requirement is softened by the fact that SHARPE recognizes the Unix line-continuation character, backslash (' \ '). Any time a line ends in a backslash, the line and its successor are viewed by SHARPE as a single line.

Chapter 2

Model Types

In this chapter, we will introduce the SHARPE model types by means of a running example. First, we will review some terminology. Then we will summarize the model types and introduce the running example. After that, each remaining section will introduce one of the model types.

2.0.1 Some Terminology

When we talk about system "reliability", we mean the probability of system failure. Reliability analysis might yield a number giving the probability that a system has failed during some given time interval, or it might yield a function *F(t)* whose value at t is the probability that the system has failed by time t.

If we introduce the possibility that a failed component in a system can be repaired, we are doing "availability" analysis; we are measuring the likelihood that a repairable system is operating at a particular time. Again, we might be seeking a number or an availability function.

The word "dependability" is used to mean "reliability" or "availability", safety, and so on [31].

When we model performance, we are generally interested in things like job completion time or system throughput. We can combine performance and "dependability" analysis by measuring performance of a system that can experience failures and possibly repair. This combined analysis is called "performability".

By a "state-space" model, we mean one where formulation of the model involves enumerating all the possible states of each component. Non-state-space models are sometimes called combinatorial models. These can describe systems concisely and have relatively efficient solution algorithms. However, the assumptions required for their solution are often not acceptable in practice. There is a tradeoff between ease of model construction and analysis on the one hand and ability to model more complex relationships between components on the other.

2.0.2 Summary of Model Types

The model types provided by SHARPE are as follows:

- reliability block diagram
  Used for: dependability analysis
  State-space? no
- fault tree
  Used for: dependability analysis
  State-space? no
- reliability graph
  Used for: dependability analysis
  State-space? no
- series-parallel directed acyclic graph
  Used for: performance analysis
  State-space? no

- product-form queueing network

  Used for: performance analysis
  State-space? no

  This model type is well-suited for capturing the affects of resource contention.

- homogeneous continuous time Markov chain

  Used for: dependability and performability analysis
  State-space? yes

  SHARPE supports acyclic, irreducible and phase-type Markov chains. SHARPE also supports Markov reward models for performability analysis; in these models, a reward rate is associated with some or all of the states.

- semi-Markov chain

  Used for: dependability and performability analysis
  State-space? yes

  SHARPE supports acyclic and irreducible semi-Markov chains.

- generalized stochastic Petri net

  Used for: dependability and performability analysis
  State-space? yes

  A GSPN is non state-space in its specification, but must be converted into a state-space model (a Markov chain) for solution.

The GSPN model type is provided in SHARPE to concisely specify large Markov models [1]. This is the only model type in SHARPE that is internally converted into another model type for solution. All the other model types are solved by the most appropriate algorithms for the model type *without* internal conversion.

2.0.3 An Example System

For our running example, we consider a a fault-tolerant, multi-processor computer with multiple memory modules. The system is able to detect a processor or memory module failure and reconfigure itself to continue operation without the failed component. We will consider two system designs:

1. all memory modules are shared by both processors.

2. some memory modules are shared and some are private to each processor. Access to private memory is faster than to shared memory.

In the remainder of this chapter, we will show how the various kinds of models SHARPE supports can be used to look at these two designs and evaluate design decisions and tradeoffs. We will present the following examples:

block diagram	analyze system reliability for first design
fault tree	analyze system reliability for second design
reliability graph	validation of fault tree model
series-parallel graph	analyze response time for one concurrent program (both designs)
queueing network	analyze system throughput (both designs)
Markov chain	analyze system availability and performability (both designs)
Petri net	validate and generalize the Markov chain model

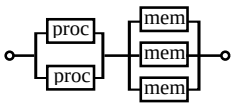


Figure 2.1: A series-parallel reliability block diagram model.

2.1 Series-Parallel Reliability Block Diagram

The SHARPE series-parallel reliability block diagram is a non-state-space model specialized for system dependability analysis. It is the usual block diagram representation of a system in which components are combined into blocks in series (all must operate for the block to operate), in parallel (one or more of the components must work for the block to operate) or in a "k-out-of-n" configuration. All of these constructs can be used together in a single block diagram.

The system structure is specified by giving a bottom-up description of series, parallel and "k-out-of-n" (k or more components must be up) combinations of components. Components of the same type that appear more than once in the system structure are assumed to be copies with independent, identical distribution functions.

Each basic component type i is assigned a number or function, usually one of the following:

- a probability Q_i, being the failure probability of the component, either time-independent or the failure probability by some particular time (the same for all components). SHARPE computes the system failure probability.
- a function $F_i(t)$, being the cumulative distribution function for the failure probability of the component. $F_i(t)$ is the probability that component i has failed by time t. SHARPE can compute the system failure distribution function and the mean and variance of the function. It can be asked to evaluate the function at particular values of t.
- a probability U_i, being the instantaneous or steady-state "unavailability" for the component. The assumption here is that there are enough independent repair facilities to repair each component that has failed; U_i is the probability that component i is not functioning. SHARPE computes the system unavailability.
- an unavailability function $U_i(t)$ where $U_i(t)$ is the instantaneous probability that component i is not functioning at time t. SHARPE computes the system unavailability function.

2.1.1 Probabilities Assigned to Components

We can use the reliability block diagram in Figure 2.1 to model the first design of our running example. This is the design where all memory modules are shared. Figure 2.1 shows the case where we have two processors and three memory modules. Suppose we are interested in finding out how system reliability is affected by the minimum number of memory modules required to continue operation.

The line numbers in Figure 2.2 are for reference only. On line 1, the keyword **block** tells SHARPE what kind of model this is. We give the model a name, *arch1*, by which we can refer to the model later, and a set of parameters. k is the number of memory modules required for the system to operate, n is the total number of memory modules, *pfail* is the processor failure probability and *mfail* is the memory failure probability. On lines 2 and 3, we define the component names and assign to each a probability value. The builtin function **prob** takes one argument, the probability that the component has failed by some implicit, unspecified time. On lines 4 through 6, we specify the structure of the block diagram, bottom up.

On lines 9 through 11, we ask for the system to be analyzed. The keyword **expr** on line 10 tells SHARPE that the rest of the line contains an arithmetic expression to be evaluated. The SHARPE builtin function **sysprob** used on line 10 provides a system failure probability (or availability) value for systems whose components were assigned probability (or availability) values. The first parameter to **sysprob** is the model name *arch1*, followed by the number of memory modules needed for operation k, the total number of memory modules (3), and the component failure probabilities: 0.0138 for a processor and 0.00692 for a memory module for an implicit time of

```
 1   block arch1(k,n,pfail,mfail)
 2   comp proc prob(pfail)
 3   comp mem prob(mfail)
 4   parallel procs proc proc
 5   kofn mems k,n,mem
 6   series top procs mems
 7   end
 8
 9   loop k,1,3,1
10      expr 1 - sysprob(arch1;k,3,.0138,.00692)
11   end
12   end
```

Figure 2.2: SHARPE Input for Block Diagram

```
k=1.000000
    1 - sysprob(arch1;k,3,.0138,.00692): 9.9981e-01

k=2.000000
    1 - sysprob(arch1;k,3,.0138,.00692): 9.9967e-01

k=3.000000
    1 - sysprob(arch1;k,3,.0138,.00692): 9.7920e-01
```

Figure 2.3: Output for Block Diagram

```
 1   block arch1(k,n,pfrate,mfrate)
 2   comp proc exp(pfrate)
 3   comp mem exp(mfrate)
 4   parallel procs proc proc
 5   kofn mems k,n,mem
 6   series top procs mems
 7   end
 8
 9   cdf (arch1;1,3,.00139,.00764)
10
11   func rel(t,k,n,pf,mf) \
12     1 - value(t;arch1;k,n,pf,mf)
13
14   loop k,1,3,1
15     expr mean(arch1;k,3,.00139,.00764)
16     expr rel(10,k,3,.00139,.00764)
17     expr rel(365,k,3,.00139,.00764)
18   end
19   end
```

Figure 2.4: Input for Block Diagram with Distribution Functions

$t = 10$ days. there are three memory modules, The variable k is to vary from 1 to 3, as indicated by the **loop** statement on line 9.

The results are shown in Figure 2.3.

If the system needs only one memory module to operate, chances are 99.8% that the system will still be up after 10 days. If two memory modules are needed, reliability is 99.7%, almost as good. But if all three memory modules are needed, reliability is down to 97.9%. It looks like it might be worth the effort to design the operating system so it can work with only two memory modules, but further effort to make it work with only one memory module probably isn't worth it.

2.1.2 Cumulative Distribution Functions Assigned to Components

Figure 2.4 shows a specification and request for analysis for the more general case where each component is assigned a distribution function rather than a probability value. It assigns the exponential distribution $F_p(t) = 1 - e^{-0.00139t}$ to processors and $F_m(t) = 1 - e^{-0.00764t}$ to memories. We note that $F_p(10) = 0.0138$ and $F_m(10) = 0.00692$, the same as the probability values assigned in Figure 2.2.

Instead of using the function **prob** to specify the component failure characteristics, we have used the builtin function **exp** on lines 2 and 3. **exp** assigns to the component the exponential distribution with one parameter, the failure rate.

On line 9, we use the keyword **cdf** to ask SHARPE to print the Cumulative Distribution Function for the system time to failure. SHARPE computes this function symbolically in the time variable t. Having the function available rather than just probability values at particular times is very valuable in modeling practice.

On lines 11 and 12, we have used the SHARPE keyword **func** to define our own function, *rel*, to be one minus the probability of failure. This makes it easier for us to look at results in terms of reliability (probabilities of being operational) rather than probabilities of failure. The parameters to *rel* are time t followed by the model parameters. The backslash character, "\", is used in SHARPE to mean line continuation.

We have used the **loop** construct to vary k between 1 and 3 and ask for the mean time to failure, the reliability at 10 days, and the reliability at 365 days. The results are shown in Figure 2.5.

CDF for system arch1: 1.0000e+00 t(0) exp(0.0000e+00 t) + -6.0000e+00 t(0) exp(-2.0833e-03 t) + 6.0000e+00 t(0) exp(-2.7778e-03 t) + 1.0000e+00 t(0) exp(-3.4722e-03 t) + -3.0000e+00 t(0) exp(-4.1667e-03 t) + 1.0000e+00 t(0) exp(-4.8611e-03 t) mean: 9.4629e+02 @ variance: 4.0922e+05	k=1.000000 mean(arch1;k,3,.00139,.00764): 9.4629 e+02 rel(10,k,3,.00139,.00764): 9.9981e-01 rel(365,k,3,.00139,.00764) : 8.3241e-01 k=2.000000 mean(arch1;k,3,.00139,.007 64): 6.9943e+02 rel(10,k,3,.00139,.00764): 9.9967e-01 rel(365,k,3,.00139,.00764) : 7.3415e-01 k=3.000000 mean(arch1;k,3,.00139,.00764): 3.7029e+02 rel(10,k,3,.00139,.00764): 9. 7920e-01 rel(365,k,3,.00139,.00764): 3 .9355e-01

Figure 2.5: Output for Block Diagram with Distribution Functions

The system failure-time distribution is $F_{sys}(t) = 1 - 6e^{-0.0021t} + 6e^{-0.0028t} + e^{-0.0035t} - 3e^{-0.0042t} + e^{-0.0049t}$. The reliability at 10 days is, as we expect, exactly the same as we found using the model in Figure 2.2 (with probability values). The mean failure times are 370 days when three memory modules are needed to run, 700 days when two are needed and 946 when only one is needed. It can be educational to use the reliability numbers to show that the mean of a distribution doesn't always give us all the important information we need to know. Suppose we consider it important that our system be able to stay up for a year at a time. We might be satisfied with a mean system failure time of 370 days (more than a year) until we look at the probability that the system is still up after a year. It is only 0.39.

2.1.3 Instantaneous Unavailability Functions Assigned To Components

When a block diagram is used to model availability, it is assumed that there enough repair resources to repair all components at the same time, if necessary. We would assign to each component an instantaneous unavailability function or a steady-state unavailability.

If the i^{th} component has exponentially distributed failure behavior with rate λ_i and repair is also exponentially distributed with rate μ_i, the component's instantaneous unavailability is

$$U_i(t) = \frac{\lambda_i}{\lambda_i + \mu_i} - \frac{\lambda_i}{\lambda_i + \mu_i} e^{-(\lambda_i + \mu_i)t} \tag{2.1}$$

and the steady-state unavailability is given by

$$\lim_{t \to \infty} U_i(t) = \frac{\lambda_i}{\lambda_i + \mu_i}$$

These expressions can be derived by solving the two-state (up/down) Markov chain for a component.

If we assign to each component in the block diagram in Figure 2.1 a function of the form of Equation 2.1, when SHARPE analyses the system the results will be a function giving the instantaneous unavailability $U_{sys}(t)$ for the system. The probabilistic "mass at infinity" $(1 - \lim_{t \to \infty} U_{sys}(t))$ of the result will be the steady-state system availability. A SHARPE input file doing just this is shown in Figure 2.6.

On lines 8 through 10, we use the keyword **poly** to define our own function (that in equation 2.1) which we will later (lines 14 and 15) assign to the block diagram's basic components. We give the function a name **U**, parameters *lambda* and *mu*, and use the keyword **gen** to tell SHARPE we are going to specify an exponential polynomial, term by term, with each term $at^k e^{bt}$ given by the comma-separated three-tuple a, k, b. The built-in function **pinf** used on line 22 computes the probabilistic "mass at infinity" of its arguments.

The results from this input file are as follows:

```
k=1.000000
```

```
 1  bind                                          13  block arch1(k,n)
 2  lambdap 1/720                                 14  comp proc U(lambdap,mup)
 3  lambdam 1/(2*720)                             15  comp mem U(lambdam,mum)
 4  mup 1/4                                       16  parallel procs proc proc
 5  mum 1/2                                       17  kofn mems k,n,mem
 6  end                                           18  series top procs mems
 7                                                19  end
 8  poly U(lambda, mu) gen \                      20
 9    lambda/(lambda + mu), 0, 0 \                21  loop k,1,3,1
10    -lambda/(lambda + mu), 0, -(lambda + mu)    22    expr 1 - pinf( arch1;k,3)
11                                                23  end
12                                                24  end
```

Figure 2.6: SHARPE Input for Availability Block Diagram

```
   1 - pinf(arch1;k,3):   3.0527e-05

k=2.000000
   1 - pinf(arch1;k,3):   3.6290e-05

k=3.000000
   1 - pinf(arch1;k,3):   4.1855e-03
```

We will refer back to these results in section 2.6.3, where we use Markov models to investigate less restrictive repair behavior.

Further examples of the use of block diagrams are given in sections 4.1 through 4.3, 6.1 and 6.2.

2.2 Fault Trees

Like the block diagram model, the fault tree model is specialized for dependability analysis. The fault tree model allows "and" gates (all inputs must fail for the gate to fail), "or" gates (any input failure causes the gate to fail), and "k-out-of-n" gates (k or more input failures cause the gate to fail). Overall inputs of the tree are known as basic events and there is a single output called the top event representing system failure event.

Each gate is defined by giving its type and identifying its inputs. The user specifies whether appearance of a event more than once in the structure is meant to be taken as the existence of a single event or as copies of a event with i.i.d. (identical, independently distributed) occurrence-time distributions.

Fault tree components are assigned probabilities or functions of the same type as those used for block diagrams. SHARPE can compute the failure-time distribution for the topmost gate, the mean and variance of the distribution, and the transient failure probability for any particular time.

Continuing with our running example, we consider the second design, the one where each processor has private memory modules and there are slower, shared memory modules. We will assume that the system will operate as long as there is at least one operational processer with access to either a private or shared memory. We cannot model this system with a block diagram, because it gives us no way to represent the way shared memories are connected to all processors and private memories to particular processors. So, we turn to a fault tree model, shown for two processors and three memory modules in Figure 2.7.

Figure 2.8 shows the model specification.

To specify fault trees, we first define basic component names and assign failure distributions to them (lines 2 through 6). The keyword **repeat** on line 6 is used to indicate that when **M3** appears more than once as an input to different gates, it is a single component whose name appears twice (in the language of fault trees, the corresponding event is said to be *shared* or *repeated*). The other possibility would be for **M3** to represent identically distributed copies of a component. On lines 7 through 11, we construct the fault tree out of **and** and **or** gates. The first word after each such gate is the name we assign to it, and the rest of the words are the inputs, which can be either basic components or the names of other gates. On lines 14 through 17, we use

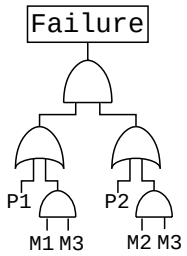


Figure 2.7: Fault Tree

```
 1   ftree t-proc-mem
 2   basic p1 exp(1/Ptime)
 3   basic p2 exp(1/Ptime)
 4   basic m1 exp(1/Mtime)
 5   basic m2 exp(1/Mtime)
 6   repeat m3 exp(1/Mtime)
 7   and mems1 m1 m3
 8   and mems2 m2 m3
 9   or pm1 p1 mems1
10   or pm2 p2 mems2
11   and top pm1 pm2
12   end
13
14   bind
15   Ptime 720
16   Mtime 2*720
17   end
18
19   cdf(t-proc-mem)
20   pqcdf(t-proc-mem)
21   end
```

Figure 2.8: SHARPE Input for Fault Tree Model

```
CDF for system t-proc-mem:

  1.0000e+00 t( 0) exp( 0.0000e+00 t)
+ -4.0000e+00 t( 0) exp(-2.0833e-03 t)
+  2.0000e+00 t( 0) exp(-2.7778e-03 t)
+  1.0000e+00 t( 0) exp(-3.4722e-03 t)
+  1.0000e+00 t( 0) exp(-4.1667e-03 t)
+ -1.0000e+00 t( 0) exp(-4.8611e-03 t)

mean: 8.7771e+02
variance: 3.5797e+05

CDF for system t-proc-mem:

[Q(p1) * Q(p2)]+
[Q(p1) * P(p2) * Q(m2) * Q(m3)]+
[P(p1) * Q(p2) * Q(m1) * Q(m3)]+
[P(p2) * Q(m1) * Q(m2) * Q(m3) * (1 - Q(p1))]
```

Figure 2.9: SHARPE Output for Fault Tree Model

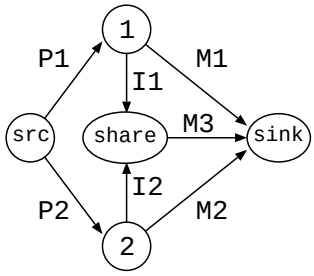


Figure 2.10: Reliability Graph

bind to assign values to the variables *Ptime* and *Mtime*. On line 19, the keyword **cdf** asks for the distribution function for the time-to-failure for the system symbolically in time t. The keyword **pqcdf** on line 20 asks for the distribution function in terms of the functions assigned to the individual components. (The name **pqcdf** comes from the fact that the distribution is printed in terms of the probabilities of failure (often denoted "p") and non-failure (often denoted "q" where q=1-p) of the individual components.) This form of the distribution can be helpful for students to check their understanding of algorithms for analyzing fault trees.

Figure 2.9 shows the results. The mean-time-to-failure is a bit worse (877) days than for the design with all shared memories, about a 7 percent difference. It is useful to have a student of modeling think about whether this result makes sense. It does, because in the system where all memories are shared, the system can operate with only one memory module no matter which particular one is the survivor. In the system with private memories, if the single survivor is a private memory attached to a processor that is not operational, the system cannot operate. However, the system may have better performance if the private memories provide faster access time. In section 2.6.2, we will explain how to use "reward rates" to help evaluate the performance-reliability tradeoff.

The **pqcdf** form of the system mean-time-to-failure can be interpreted term by term: the first represents the case that both processors have failed, the second and third the case that exactly one processor is up, but both the shared memory and its private memory have failed, and the fourth the case that both processors are up, but all memory modules have failed.

Experiments with SHARPE can help illustrate the difference between independently, identically distributed copies of a component and a component that is physically shared. We recall that the event **M3** is shared by two gates in the fault tree of Figure 2.7 and that the interpretation we have put on this duplication is that there is one physical component whose failure can affect both of two subsystems. It is also possible to interpret the duplication as the presence of two physically different components with the same (but statistically independent) time-to-failure distributions. It would be easy to make a mistake when using a modeling program and have the program take the opposite interpretation than the one we wanted. (In the case of SHARPE, this would happen if we had used the word **basic** instead of **repeat** on the line 6 in Figure 2.8.) If we had made that mistake, we would have found a mean time to failure of 882 days instead of 877.

One way to help detect this kind of specification error is to validate the model before using its results. With the help of SHARPE, we can validate the fault tree model by constructing and analyzing a different, but equivalent, model: the reliability graph to be discussed in the next section.

Further examples of use of fault trees are given in sections 4.5 through 4.7, 6.11 and 6.12.

2.3 Reliability Graphs

The reliability graph model consists of a set of nodes and edges, where the edges represent components that may fail. The graph contains one node, the "source", with no incoming edges and one node, the "sink" with no outgoing edges. A system represented by a reliability graph fails when there is no path from source to sink.

The edges can be assigned failure or unavailability values or functions, the same as is done for fault trees and reliability block diagrams.

Referring back to our running example, we can validate the fault tree of Figure 2.7 with the reliability graph shown in Figure 4.4.

In this particular model, processor failures happen along the edges labeled *P1* and *P2* and memory failures

```
1   * reliability graph for
2   * 2-processor,
3   * 3-memory system
4
5   relgraph rel-proc-mem
6   src P1 exp(1/Ptime)
7   src P2 exp(1/Ptime)
8   P1 sink exp(1/Mtime)
9   P2 sink exp(1/Mtime)
10  P1 share inf
11  P2 share inf
12  share sink exp(1/Mtime)
13  end
14  bind
15  Ptime 720
16  Mtime 2*720
17  end
18
19  cdf(rel-proc-mem)
20
21  end
22
23
24
25
26
```

Figure 2.11: SHARPE Input for Reliability Graph

happen along the edges *M1*, *M2* and *M3*. The edges *I1* and *I2* do not represent system components; they represent the structural nature of the system (that *M3* is shared). SHARPE gives us a way of saying that edges *I1* and *I2* cannot fail; we assign the "infinite" distribution, defined by $I(t) = 0$, to them. There is a path from source to sink if *P1* and *M1* are up or if *P1* and *M3* are up, and similarly for paths involving *P2*.

Figure 2.11 shows the SHARPE input file for the reliability graph. Lines starting with "*" are comments, ignored by SHARPE. We specify each edge in the graph (on lines 6 through 12) by giving two vertex names followed by a distribution function. The distribution **inf** (for the edges *I1* and *I2* on lines 10 and 11) is built into SHARPE.

The failure-time distribution function computed in response to the keyword **cdf** on line 19 will be the same as that computed for the fault tree in the previous section.

For more examples of reliability graph models, see section 6.2.

2.4 Series-Parallel Graphs

The series-parallel graph model consists of a series-parallel directed acyclic graph, with the nodes of the graph representing activities and the edges representing a precedence relation imposed on the activities. All activities are assumed to be statistically independent, and there is no restriction on the degree of concurrency of activities.

2.4.1 A Graph Example

Before giving a detailed description of this model type, let us consider our example system again. So far we have been looking at system reliability. Now suppose we would like to investigate system performance. We consider the running time of a program that has components that can be executed in parallel. The task graph labeled **a** in Figure 2.12 models a very simple parallel program. It starts with an initialization section (A), then performs a long calculation in two parts, which may be done in parallel (B and C), then ends with some final processing (D). Program sections B, C and D require access to memory. The profile of the memory accesses depends on whether there are private memory modules and which memory modules are operational.

For a system with all shared memory, all memory accesses are obviously to shared memory. If the system has private memory modules, sections B and C use private memory during their calculations, then deposit results in shared memory for use by section D, which also accesses shared memory.

Graph **b** in Figure 2.12 represents the execution of the program in a system having all shared memories. We have added tasks smB, smC and smD to represent the memory accesses.

Graph **c** in Figure 2.12 represents the execution of the program in a system having the second design, with two private memory modules. In this case, there are two memory-access tasks for each of the two parallel parts of the program, the first to private memory and the second to shared (to deposit results). (We will explain the meaning of the bold circle in section 2.6.2).

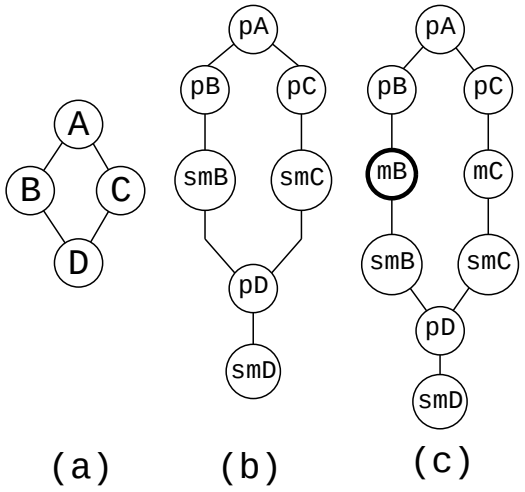


Figure 2.12: Graph Models

Figure 2.13 shows how the graph in Figure 2.12(b) would be specified for SHARPE. On line 1, we specify the model type (**graph**) and give it a name (*parallel-1*) and parameters which are the service rates for tasks *smB*, *smC* and *smD*. We chose to parameterize the service rates for these particular tasks because we plan to use these models again in section 2.6.2, where we will want to vary the rates for these tasks.

On lines 2 through 8 we specify the precedence restrictions on the tasks. On line 11, we use the keywords **exit** and **max** to indicate that the two subgraphs that follow *pA*, the first consisting of *pB* followed by *smB* and the second consisting of *pC* followed by *smC*, must both finish before task *pD* can begin. SHARPE also allows exit types **min**, meaning that a set of subgraphs is finished when the first subgraph is finished, **kofn**, meaning the set is finished when k out of n subgraphs are finished, and **prob**, meaning that only one of the set of subgraphs is chosen to be executed.

On lines 11 through 17, we assign to each task a cumulative distribution function for the time it takes the task to be completed. In this case we have used the exponential distribution; more general distributions are allowed. On lines 20 through 27, we assign values to the variables used. On lines 29 and 30, we assign to the variable *G32* the average rate at which the system can finish jobs like this, working one at a time (this is the inverse of the mean time it takes to finish one job). On line 31, we ask to have this rate printed.

2.4.2 Description of the Graph Model

Series-parallel graphs are "well-structured". They are built by starting with single nodes and combining the single nodes either in series or in parallel. The subgraphs obtained in that way may then be recursively combined in series or in parallel with other activities or subgraphs. An important characteristic of this structure is that whenever multiple edges leave a node, they lead to two or more *disjoint* parallel subgraphs. Once an activity with multiple successor activities is finished, the disjoint subgraphs that follow it proceed in parallel, with neither having any precedence dependencies on what is happening in the other subgraph.

Figure 2.14 contains some examples that may help in understanding when a graph is series-parallel and when it is not.

In this and many subsequent pictures of graphs, the direction of the edges is not shown explicitly; when direction is not shown, it is assumed that all edges point downward. Graph 1 is series-parallel. Graphs 2 and 3 are not series-parallel, and any graph that contains either of these graphs as a subgraph is not series-parallel. Graph 4 is not series-parallel because of the redundant edge from node A to node C. Graph 5 is series-parallel and is equivalent to Graph 4 if node D represents an activity that takes no time. Graph 6 is not series-parallel but can be written equivalently as Graph 7 if node 5 takes no time.

When a graph consists of two series subgraphs, the second subgraph begins when the first has finished. If a graph is composed of parallel subgraphs, we allow several interpretations of what it means for the graph to be "finished".

```
graph parallel-1 (mBrate,mCrate,mDrate)
pA pB
pA pC
pB mB
mB pD
pC mC
mC pD
pD mD
end
exit pA max
dist pA exp(pArate)
dist pB exp(pBrate)
dist pC exp(pCrate)
dist pD exp(pDrate)
dist mB exp(mBrate)
dist mC exp(mCrate)
dist mD exp(mDrate)
end

bind
smrate 1800
smrateD 9600
pArate 7200
pBrate 60
pCrate 50
pDrate 3600
end

var G32 1/mean(parallel-1; \
    smrate,smrate,smrateD)
expr G32

end
```

Figure 2.13: Input for Parallel Graph Model

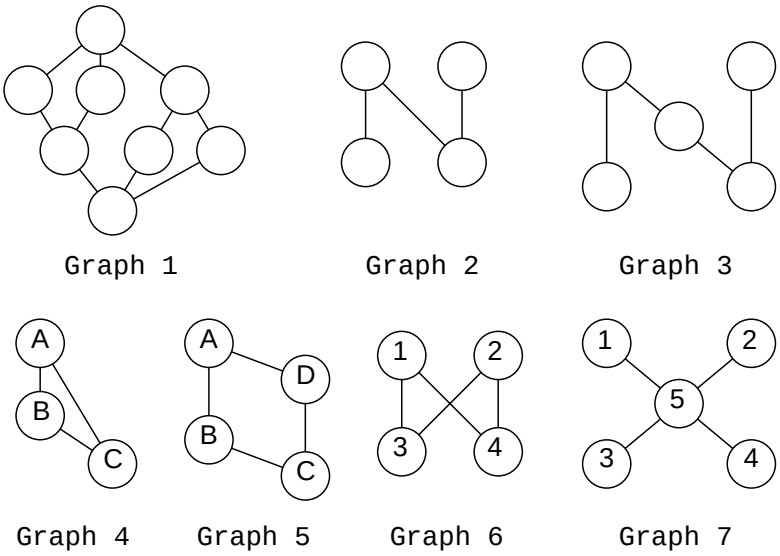


Figure 2.14: Some Graph Examples

1. probabilistic

   Only one of the parallel subgraphs is chosen. The finishing time is the probabilistic sum of the finishing times of the parallel subgraphs. Such probabilistic branching can be used to model conditional statements in programs.

2. maximum

   All of the parallel subgraphs run concurrently. The finishing time is the time when the last subgraph finishes. Maximum type branching would be used to model a parallel algorithm which requires that all subtasks finish before the answer is known. Maximum parallelism also models the time-to-failure of a parallel combination of components, since failure does not occur until all the components have failed.

3. minimum

   All of the parallel subgraphs run concurrently. The finishing time is the time when the first subgraph finishes. We do not care whether the remaining subgraphs run to completion or are aborted. Minimum parallel subgraphs can model the parallel execution of a nondeterministic algorithm (several algorithms are run simultaneously, and the first one to finish provides the desired answer). Minimum parallelism also models the time-to-failure of a series combination of components, since failure occurs when the first component fails. Another application is while carrying a parallel search of a database; soon as one of concurrent tasks finds the required item, the overall search terminates.

4. k-out-of-n

   All of the parallel subgraphs run concurrently. The finishing time is the time when the k^{th} subgraph finishes. Note that "maximum" and "minimum" are special cases of k-out-of-n. k-out-of-n parallelism can be used to model systems containing redundant components, where a minimum number of components must be operating.

It is important to note that the use of the words "series" and "parallel" is quite different when talking about series-parallel graphs than when talking about series-parallel reliability block diagrams. In the block diagram world, "series" means that components are combined in such a way that all components must work in order for their combination to work; "parallel" means that components are replicated and some minimum number of them must work in order for their combination to work. In the graph world, "parallel" means that activities happen concurrently. "Block-diagram-series" and "block-diagram-parallel/active spares" are both special cases of "graph-parallel", since in these constructs all components are functioning, and the process of failure is going on in all components at the same time. "Graph-series" means that activities happen sequentially. In the block diagram world, sequential activities occur when there is "block-diagram-parallel" redundancy with cold spares.

Examples of the use of this model type are given in sections 5. For more examples of graph models, see sections 5.1.1 through 5.1.5, 6.3 and 6.4.

2.5 Product-Form Queueing Networks

A queueing network is a collection of "service centers", each containing one or more servers and a queue to hold "jobs" that require service. SHARPE supports a subset of those queueing networks having what is called "product-form" solution [53]. It is further assumed that the network is a closed queueing network. This means that when a job finishes service, it moves to another service center (it cannot leave the network). The distribution functions for the service times are exponential. The service types (order in which jobs are chosen for service, degree of server sharing, possible use of preemption) are chosen from a set of possibilities that allow for product-form solution; the possibilities are listed in section A.4.6 in Appendix A.

A queueing network is specified by listing the service centers, the transition probabilities for jobs going from one service center to another, the service characteristics and distributions for the service centers, and the number of jobs in the network.

SHARPE allows "multi-chain" queueing networks, in which there are several classes, or "chains" of jobs. Each chain of jobs has its own set of transition probabilities (from service center to service center), and the service centers may have different service time distributions (but not different service types) for the different chains.

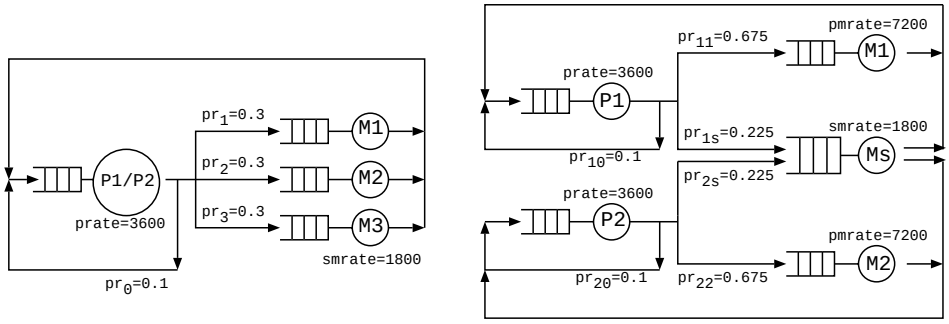


Figure 2.15: Queueing Network Models for the Two System Designs

SHARPE can compute steady-steady throughput, utilization, average response time and average queue length for each service center. For multi-chain networks, SHARPE can compute these measures on a per-chain basis.

In the previous section, we analyzed system performance from the point of view of one job that had the system all to itself. If we want to look at overall system performance, we can use queueing network models to capture the effects of contention for the processor and memory resources in the system.

We assume the memory modules are servers in the sense that they queue requests and perform block transfers. To set up a realistic queueing model, we would have to take into account the proposed operating system design, especially the scheduling aspects, and we would need some kind of expected workload characterization. For the sake of illustration, we will use the closed queueing network models shown in Figure 2.15.

The system on the left is for the design containing two processors and three shared memory modules. We model the two processors by a multiple-server station. That is, jobs wait in a single queue and enter whichever server becomes free. We assume that when a job requires access to memory, we know the likelihood that it will require each particular memory module, pr_i for module Mi. We also assume that a job finishes after some number of visits to the processor; pr_0 is the probability that a job is finished when it leaves one of the processors. As is usual for closed queueing networks, the assumption is that each finished job is replaced by a statistically identical new job.

The system on the right is for the design containing two private memory modules. For this system, we assume that jobs are targeted to particular processors. This is reasonable, since once a job starts on a processor, we want it to continue there where it has access to that processor's private memory. We carry out this assumption by making the queueing network a "multiple-chain" queueing network, in this case having two "chains", or classes of jobs. Jobs in the first class go from *P1* to either *M1* or *Ms* and back to *P1* and jobs in the second class go from *P2* to either *M2* or *Ms* and back to *P2*.

To model the systems when one memory has failed, we would remove the server *M1* (and its queue) from each of the models and adjust the probabilities pr_i and pr_{ij} appropriately.

Figure 2.16 shows the SHARPE specification for the two networks in Figure 2.15. The first specification (lines 14 through 31) is for the single-class queueing network (keyword **pfqn**), called *arch1-32*. The specification has three sections, each ending with the keyword **end**. First (lines 16 through 22) we give a series of triples, where the first two elements are stations and the third is the probability that a job visits the second station after completing service at the first. We do not have to give explicitly the new job path from the processor station back to itself. The second part of the specification (lines 24 through 27) assigns the service characteristics of the stations. Station *P1/P2* is a "multiple server" (*ms*) composed of 2 servers, each with exponential service time distribution with para meter *prate*. Stations *M1*, *M2* and *M3* are "first-come-first-server" (**fcfs**) servers with rate *smrate*. The third section (line 30) gives the number of customers (c) in the network. The word *customers* preceding c is just a placeholder; in a multiple-chain queueing network it would give the name of a chain. We define the variable *P32* to be the system throughput (the rate at which jobs go along the path from *P1/P2* back to *P1/P2*.

The second specification is for the multiple-class queueing network (keyword **mpfqn**). The first section (lines 43 through 55) now has a subsection showing the shape of the network for each chain. Each subsection starts with a line with the keyword **chain** and the name of the chain on it. The second section (lines 56 through 66) again gives the service characteristics for each station. Although we have not used the feature here, it is possible

```
bind
c 6
c1 3
c2 3
p0 .1
prate 3600
pmrate 7200
smrate 1800
end

* queueing network for 3-memory
* two-processor system, design 1

pfqn arch1-32
* section 1: network shape
P1/P2 M1 (1-p0)/3
P1/P2 M2 (1-p0)/3
P1/P2 M3 (1-p0)/3
M1 P1/P2 1
M2 P1/P2 1
M3 P1/P2 1
end
* section 2: station types
P1/P2 ms 2,prate
M1 fcfs smrate
M2 fcfs smrate
M3 fcfs smrate
end
* section 3: # customers
customers c
end

var P32 2*prate*util(arch1-32,P1/P2)*p0
```

```
* queueing network for 3-memory
* two-processor system, design 2

mpfqn arch2-11111
chain 1
P1 M1 .75*(1-p0)
P1 Ms .25*(1-p0)
M1 P1 1
Ms P1 1
end
chain 2
P2 M2 .75*(1-p0)
P2 Ms .25*(1-p0)
M2 P2 1
Ms P2 1
end
end
P1 fcfs prate
end
P2 fcfs prate
end
M1 fcfs pmrate
end
M2 fcfs pmrate
end
Ms fcfs smrate
end
end
1 c1
2 c2
end

var P11111 2*prate*mutil(arch2-11111,P1,1)*p0

expr P32, P11111

end
```

Figure 2.16: SHARPE Input for Queueing Models

to give different service rates for each chain. That is why each service characteristic specification ends with **end**. In the third section (lines 67 and 68) we give the number of customers for each chain.

The results are $P32 = 526.7$ and $P11111 = 596.7$. As expected, the system with private memories provides higher system throughput.

For more examples of queueing network models, see sections 5.2.1, 5.2.3 and 6.6.

2.6 Markov and Markov Reward Models

The Markov model is a finite state, homogeneous, continuous-time Markov chain. SHARPE allows three types of Markov chains:

1. **acyclic**. A chain is acyclic if every state is visited at most once (the graph of the chain contains no cycles).

2. **irreducible**. A chain is irreducible if every state can be reached from every other state For an irreducible chain, a steady-state probability of being in each state exists and is independent of the initial probabilities [53].

3. **phase-type**. A phase-type chain is a chain that is neither acyclic nor irreducible It has one or more absorbing states and one or more transient states. Thus, it is guaranteed that an absorbing state will eventually be reached.

The structure of a Markov chain is specified to SHARPE by giving each possible state transition with its associated instantaneous transition rate. Optionally, each state may be assigned a reward rate. If the chain is not irreducible, initial state probabilities must be supplied. If the chain is irreducible, initial state probabilities are not needed unless SHARPE will be asked to compute transient measures. For backward compatibility with earlier versions of SHARPE, it is assumed that initial state probabilities will not be supplied unless a special indication is given that the specification will include them. See section 7.2 for an example.

SHARPE can compute the following functions and probabilities:

- distribution function for time to reach a particular absorbing state (except for irreducible chains).

- distribution function for time to reach absorption regardless of which particular absorbing state it is (except for irreducible chains).

- transient probability function with parameter t, giving the probability of being in a particular state at time t.

- transient probability (not a function but a numerical value) of being in a state at a particular time.

- probability of ever visiting a particular state (except for irreducible chains, for which this probability is 1 for every state).

- steady-state probability of being in a particular state (only for irreducible chains).

- distribution function for the time until a particular state transition occurs (only for acyclic chains).

- expected reward rate at a particular time.

- expected cumulative reward by a particular time.

- reward function with parameter r, giving the probability that cumulative reward at time of absorption is less than or equal to r (except for irreducible chains).

- steady-state expected reward rate (only for irreducible chains).

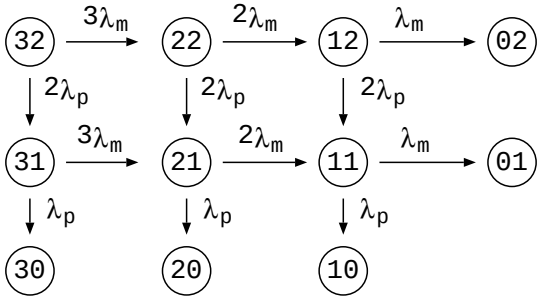


Figure 2.17: Markov Model of Shared-memory System

To illustrate the use of Markov chain models, we return to our running example. We'll start by modeling system reliability using an acyclic Markov chain, then extend the model in two different ways, first to an acyclic Markov reward model to do performability analysis, then to an irreducible Markov chain to analyze system availability.

For examples of phase-type Markov chains, see sections 4.10, 6.4 and 6.10. For more examples of acyclic Markov chains see sections 4.8, 4.9, 6.1 and 6.3. For more examples of irreducible Markov chains, see sections 4.11, 5.2.4, 5.2.5 and 6.11.

It is interesting to note that SHARPE can solve for steady-state measures of finite, homogeneous, discrete-time Markov chains. An example will be given in section **??**.

2.6.1 Acyclic Markov Model Used For Reliability Analysis

We start by constructing a Markov model equivalent of the reliability block diagram in Figure 2.1. This is a model of the system where all memory modules are shared. The acyclic Markov model is shown in Figure 2.17. When the system is in state **mp**, **m** memory modules and **p** processors are working. The failure rates are λ_m for a memory module and λ_p for a processor.

The distribution function for the time-to-absorption in this acyclic Markov chain is the time-to-failure distribution for the system. It is the same distribution function as obtained by analyzing the the block diagram model in Figure 2.7.

To see why a block diagram is a non-state-space model and a Markov chain is a state-space model, consider what you would have to do to modify the models in Figures 2.1 and 2.17 so that there were four memory modules rather than three. For the block diagram model, all that is needed is to add one more memory module to the stack of three. In fact, the SHARPE specification for the model would not change at all. We would simply analyze the model with $n = 4$ instead of $n = 3$. To add a memory module to the Markov model, we would have to add another column to the model; we would be adding three states and five transitions.

We can also construct a Markov chain for the second design (one shared memory, two private memories) we modeled before using the fault tree in Figure 2.7. In this chain (not shown), each state would be a five-tuple of binary digits ($p_1 p_2 - m_1 m_2 - m_s$), one for each of the processors and memories. In state **10-10-1**, processor 2 and its private memory have failed. The chain has twenty states. Because this time there really was some risk that we would make a mistake when setting up the model, making a validity check against the results of the fault tree model from Figure 2.7 would have more than just instructional value.

This second Markov chain increases in size even faster than the first when we add processor or memory components, while the fault tree would gain just one or two components (two for the case of shared memory modules) for each new component.

It would seem that it would always be to our advantage to use the more efficient reliability block diagram and fault tree models. But, although it is easy to extend the block diagram to include another independent component, we cannot extend it to include consideration of system performance (for the reliability case) or shared repair of components (for the availability case). In the next two sections, we show how this extension is done using Markov models.

2.6.2 Acyclic Markov Reward Model Used For Performability Analysis

Now we are in a position to extend the two Markov models to take performance into account. We can attach a "reward rate" to each state, making the model a "Markov reward" model. A "reward rate" is some measure of the "goodness" of the state, possibly a performance measure. A Markov reward model can be analyzed for measures like expected cumulative reward by time of absorption (failure in this case) and the function $R(r)$ defined to be the probability that the cumulative reward is less than or equal to r by time of absorption. So, we can use this model to analyze both aspects of our design tradeoff: greater reward rate for shorter time versus smaller reward rate for more time.

As yet another validity check, we start by assigning reward rate 1 to each of the non-failure states. When we do this, "cumulative reward by time of failure" is really the same as "accumulated time by time of failure", so we expect (and find) $R(r)$ to be the same as $F(t)$, the failure-time distribution function.

We can compute the reward rates from performance models of the system, such as those in sections 2.4 and 2.5. It is important to choose the performance model parameters carefully, so that the time units are compatible with the time units used for failure rates in the Markov chains.

If we use queueing models, our reward rates capture performance from a system perspective. For each of the states in the two Markov chains of section 2.6.1, the one in Figure 2.17 and the one not shown having each state represented by a five-tuple, we constructed a queueing model.

The two models in Figure 2.15 are for state **32** of the first Markov chain and state **11-11-1** of the second chain.

If we use graph models to get our reward rates, the rates capture performance from the point of view of a single parallel program. We constructed graph models for each of the states in the two Markov chains (some could serve for more than one state). Analysis of graph **b** in Figure 2.12 gives us a reward rate for state **32** of the first Markov chain. Analysis of graph **c** gives us a reward rate for state **11-11-1** in the second chain if we use a service rate for a private memory for the task in the bold circle, and a reward rate for states **10-11-1** and **11-10-1** if we use a shared memory service rate. The following tables show the reward rates for the two designs using the two different performance models.

1st design

state	reward rate, queueing model	reward rate, graph model
32	526.7	34.89
22	454.3	34.89
12	313.0	34.89
31	157.1	26.11
21	139.5	26.11
11	93.7	26.11

2nd design

P1P2-M1M2-M3	reward rate, queueing model	reward rate, graph model
11-11-1	596.7	35.28
10-11-1	177.5	26.72
11-01-1	389.8	35.04
11-11-0	338.7	26.72
10-10-1	177.5	26.72
10-11-0	178.9	26.72
10-01-1	93.8	26.11
11-00-1	313.0	34.76

Reward rates from the queueing networks are in number of job completions per hour. Reward rates from the graph models are the inverse of the mean-time-to-completion for one program consisting of the tasks in the graph. Again, we can't compare these reward rates directly, but we can compare the two design choices for percentage difference in accumulated reward by system failure.

We show part of the SHARPE hierarchical model specification in Figure 2.18. Lines 5 through 22 show the specification for the single-chain queueing network *arch1-32* that models the all-shared-memory system with all

```
* queueing network for state 11
* of first architecture (other
* networks not shown)

pfqn arch1-32
* section 1: network shape
P1/P2 M1 (1-p0)/3
P1/P2 M2 (1-p0)/3
P1/P2 M3 (1-p0)/3
M1 P1/P2 1
M2 P1/P2 1
M3 P1/P2 1
end
* section 2: station types
P1/P2 ms 2,prate
M1 fcfs smrate
M2 fcfs smrate
M3 fcfs smrate
end
* section 3: # customers
customers c
end

* define variables that give the expected
* job completion rate (only one shown)

var P11 prate*util(arch1-11,proc1)*p0/2

* graph for state 32 of second arch
* (other graphs not shown)

graph par-2 (mBrate,mCrate,mDrate)
pA pB
pA pC
pB mB
mB smB
smB pD
pC mC
mC smC
smC pD
pD mD
end
exit pA max
dist pA exp(pArate)
dist pB exp(pBrate)
dist pC exp(pCrate)
dist pD exp(pDrate)
dist mB exp(mBrate)
dist smB exp(smrate)
dist mC exp(mCrate)
dist smC exp(smrate)
dist mD exp(mDrate)
end

* define variables that give the expected
* jobs per hour (only one shown)

var G11111 \
    1/mean(par-2;pmrate,pmrate,smrate)


* Markov reward model for first architecture

markov arch1(r32,r22,r12,r31,r21,r11)
32 22 3*lambdam
22 12 2*lambdam
12 Fm lambdam
32 31 2*lambdap
31 Fp lambdap
22 21 2*lambdap
21 Fp lambdap
12 11 2*lambdap
11 Fp lambdap
31 21 3*lambdam
21 11 2*lambdam
11 Fm lambdam
reward
32 r32
22 r22
12 r12
31 r31
21 r21
11 r11
end
32 1.0
end

* Markov reward model for second architecture

markov arch1(r11111, r10111, r11011, r11110,\
    r10101, r10110, r10011, r11001)
.
.
.
end

* define percentage by which performability
* of 1st design is better than second

func percent(r32,r22,r12,r31,r21,r11,\
    r11111, r10111, r11011, r11110,\
    r10101, r10110, r10011, r11001) \
   (rmean (arch1;r32,r22,r12,r31,r21,r11) - \
    rmean(arch2;r11111, r10111, r11011, \
    r11110, r10101, r10110, r10011, r11001)) / \
   rmean (arch1;r32,r22,r12,r31,r21,r11)

echo PERCENT IMPROVEMENT OF ARCH 1 OVER ARCH 2
echo USING PFQNS FOR REWARDS

expr percent(P32,P22,P12,P31,P21,P11,\
    P11111, P10111, P11011, P11110,\
   P10101, P10110, P10011, P11001)

echo PERCENT IMPROVEMENT OF ARCH 1 OVER ARCH 2
echo USING GRAPHS FOR REWARDS

expr percent(G32,G22,G12,G31,G21,G11,\
    G11111, G10111, G11011, G11110,\
    G10101, G10110, G10011, G11001)

end
```

Figure 2.18: SHARPE input for Markov reward model

components functioning. On line 27, we define the variable *P11* to be the network throughput (rate at which jobs travel along the "new job" path. We use the SHARPE built-in function **util**, which gives the service station utilization in a queueing network. If this were the complete input file, it would include specifications for all of the queueing networks (for both system designs and all combinations of number of functioning components) and definitions of all of the variables *Pij* giving the system throughput for systems of the first design with i memory modules and j processors functioning and all of the variables *Pnnnnn* giving the system throughput for systems of the second design where each n corresponds to one of the five components and is 1 or 0 depending on whether the component is functioning or not.

Lines 32 through 53 specify the graph model (**graph** *par-2*) for the private-memory design with all components functioning (see Figure 2.12(c)). On lines 58 and 59, we define the expected number of jobs completed per hour, *G11111*, for this system. It is the inverse of the mean completion time of a job, as given by the SHARPE built-in function **mean** with appropriate parameters. If this were the complete input file, it would include specifications for all of the graph models for both system designs and all combinations of functioning components and definitions of the variables *Gij* and *Gnnnnn*.

On lines 64 through 86, we specify the Markov model **arch1** for the all-shared-memory system. On line 64, we assign the model the name *arch1* and six parameters, which we will use as the reward rates for the six operational states of the system. On lines 65 through 76 we specify the chain's state transitions, one per line, each transition accompanied by the transition rate. The keyword **reward** on line 77 tells SHARPE we wish to assign reward rates to the states (reward rates are optional in SHARPE Markov models). On lines 78 through 83, we assign reward rates. By default, if no reward rate is assigned to a state, SHARPE assigns zero. The last section of the Markov chain specification (line 85) specifies initial state probabilities. The complete input file would also include the Markov chain specification for the second design, called *arch2*.

On lines 100 through 106, we define the function *percent* to be $\frac{\mathbf{rmean}(arch1;\ldots)}{\mathbf{rmean}(arch1;\ldots)-\mathbf{rmean}(arch2;\ldots)}$, where **mean** is a built-in SHARPE functioning giving the expected accumulated reward for a Markov chain and given arguments. *percent* tells us the relative improvement of the first design over the second. The arguments to *percent* are passed along via **rmean** to the Markov chains to be used as reward rates.

The SHARPE keyword **echo** causes the remainder of the line on which it appears to be echoed to output and otherwise ignored.

On lines 111 through 113, we evaluate the function *percent* with arguments (reward rates) being the throughput values from the queueing network performance models. On lines 118 through 120, we evaluate *percent* with arguments being the job completion rates from the graph performance models.

This is the first time we have shown SHARPE being used to combine models hierarchically. We have used three different model types: queueing network models and graph models "inside" a Markov model.

We found that, using the queueing network models to calculate reward rates, the expected number of jobs completed by the time the system fails is 24,467 for the first design and 25,217 for the second. The second design is 3.1% better. Using the graph models to calculate the reward rates, the first design is 7.3% better.

2.6.3 Irreducible Markov Model Used For Availability Analysis

In this section, we will show how we can use an irreducible Markov model to investigate system behavior when failed components can be repaired or replaced. We will calculate the "availability" of the system, by which we mean the probability (whether instantaneous or steady-state) that the system is functioning at a given time. We will examine the all-shared-memory system, and we will look at three repair strategies:

1. There are enough repair resources to repair all components at the same time, if necessary.

2. There are two repair facilities, one for processors and one for memory modules, each able to handle one component at a time.

3. There is one repair facility, able to handle one component at a time. Processor repair gets priority over memory repair.

We looked at the first repair strategy in section 2.1.3, where we used a reliability block diagram.

To deal with the second and third repair strategies, we cannot use the block diagram model. The block diagram assumes that all components are statistically independent, but if components must share repair facilities

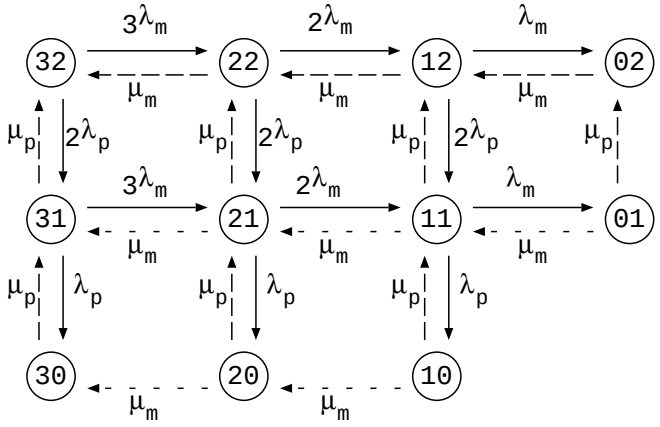


Figure 2.19: Markov chain model of 2-memory, 3-processor system

their behavior (with respect to being operational or failed) is not independent; it depends on whether or not other components are in a failure state. If the failure and repair distributions are exponential, we can use a Markov chain model.

Consider the Markov model in Figure 2.19. The state name *mp* represents the system when *m* memory units are functional and *p* processors are functional. The model with all of the solid and dashed-line transitions is for the second repair strategy (one repair facility for processors and one for memories). To apply the model to the third strategy (one repair facility for everything), we exclude the lines with smaller dashes; these represent memory repair in the case where there has been a processor failure.

Figure 2.20 shows how we can specify this model for SHARPE. On lines 2 through 5, we bind values to the failure rates *lambdap* and *lambdam* and repair rates *mup* and *mum*. On lines 8 through 43, we specify the Markov chain state transitions and transition rates. Note that on lines 38 through 42 we assign the rates *mum*onerepair*. If *onerepair* is 0, this effectively removes the smaller dashed lines from Figure 2.19. If *onerepair* is 1, the rates on the smaller dashed lines are *mum*.

The built-in function **prob**, when given as arguments an irreducible Markov chain and a state in the chain, evaluates to the steady-state probability of being in the state. On lines 45 and 46 of Figure 2.20, we define the variable *munavail1* to be the steady-state probability of system failure if we need only one memory module for the system to operate; it is the sum of the steady-state probabilities for the states where all processors or all memories have failed. On line 47, we define *munavail2* to be the steady-state probability of system failure if we need two memory modules; it is the sum of *munavail1* and the steady-state probabilities for the states where there is at least one processor and exactly one memory module functioning. Similarly, *munavail3*, the steady-state failure probability if we need all three memory modules, is defined in terms of *munavail2*. On lines 53 and 54, we set *onerepair* to 1 and ask for the steady-state system probabilities for one repair facility per component type. On lines 57 and 58, we set *onerepair* to 0 and ask for the steady-state system probabilities for one repair facility for all components.

The following table shows steady-state unavailability for the three repair strategies for the cases where one, two and three memory modules are needed for the system to remain operational.

# memories needed	steady-state unavailability		
	unlimited	one repair per component type	one repair
1	.0000305	.0000611	.0000611
2	.0000363	.0000726	.0000738
3	.00418	.00422	.00436

Under all three repair strategies, if the system needs two memory modules to keep working the steady-state unavailability is very little worse than if only one would suffice; but if three memory modules are needed, then the steady-state unavailability is considerably higher, but still very small.

```
 1  bind
 2  lambdap 1/720
 3  lambdam 1/(2*720)
 4  mup 1/4
 5  mum 1/2
 6  end
 7
 8  markov M
 9  * memory failure
10  32 22 3*lambdam
11  22 12 2*lambdam
12  12 02 lambdam
13  31 21 3*lambdam
14  21 11 2*lambdam
15  11 01 lambdam
16
17  * processor failure
18  32 31 2*lambdap
19  31 30 lambdap
20  22 21 2*lambdap
21  21 20 lambdap
22  12 11 2*lambdap
23  11 10 lambdap
24
25  * processor repair
26  30 31 mup
27  31 32 mup
28  20 21 mup
29  21 22 mup
30  10 11 mup
31  11 12 mup
32  01 02 mup
33
34  * memory repair
35  22 32 mum
36  12 22 mum
37  02 12 mum
38  21 31 mum*onerepair
39  11 21 mum*onerepair
40  01 11 mum*onerepair
41  20 30 mum*onerepair
42  10 20 mum*onerepair
43  end
44
45  var munavail1 prob(M,30)+prob(M,20)+ \
46     prob(M,10)+prob(M,02)+prob(M,01)
47  var munavail2 munavail1 +\
48     prob(M,11) + prob(M,12)
49  var munavail3 munavail2 +\
50     prob(M,21) + prob(M,22)
51
52  * one repair per component type
53  bind onerepair 1
54  expr munavail1, munavail2, munavail3
55
56  * one repair facility
57  bind onerepair 0
58  expr munavail1, munavail2, munavail3
59
60  end
61
```

Figure 2.20: SHARPE Input for Markov Availability Model

We also observed that for a particular number of required memories, the repair strategy didn't seem to make much difference in the system availability especially between the second and third repair strategies. In a teaching situation, we could advise students to use the SHARPE **prob** function to confirm that this was a reasonable answer by looking at the steady-state probabilities for each state of the Markov chain. For the single-repair case, the probabilities are as follows:

one kind of component has failed		both kinds of components have failed	
state	steady-state probability	state	steady-state probability
32	0.984	21	0.00013
31	0.0108	20	0.000007
30	0.00006	11	0.000009
22	0.00415	10	0.000000005
12	0.000119	01	0.000000002
02	0.00000002		

The probability of being in a state where both processors and memories have failed (states *21*, *11*, *01*, *20* and *10*) is so small that it isn't surprising that the difference in repair rates while in those states doesn't have much of an effect on system unavailability. We would expect that if repair rates were much slower in comparison to

failure rates, that we would see much more of a difference in unavailability for the different repair schemes (in addition, of course, to higher unavailability for all the schemes). In fact, we tried this and the results were as expected.

We note that we could have used the Markov model for the first repair strategy also. We would have assigned different transition rates to the repair transitions to reflect the fact that more than one component can be repaired at a time. As an example, the rate for the transition *02* to *12* would be $3 * \mu_m$ rather than μ_m. We used SHARPE to verify that the analysis of this model yields the same results as the block diagram model. It can be especially satisfying for students to validate the models in this way, since the model types and analysis algorithms are completely different. We note again that the block diagram model is not only much easier to construct, but much more efficient to analyze.

2.7 Semi-Markov Models

The semi-Markov model is a discrete-state, continuous-time, acyclic or irreducible semi-Markov chain. Phase-type semi-Markov chains are not supported.

The only difference between a semi-Markov model and an acyclic or irreducible Markov model is in the specification of distributions on the edges. In a Markov chain, the rate associated with each state transition must be constant; that is, the distribution for the time between entering the source state and entering the destination state is the exponential distribution. Because the rate is constant, for a Markov chain the behavior of a transition can be specified completely by just giving the rate.

For a semi-Markov chain, the transition rates leading out of a state may depend upon the time already spent in that state, and hence the distribution for the state's holding time can be arbitrary. SHARPE expects users to specify an exponomial distribution for each edge in a semi-Markov chain.

All state transition distributions are either conditional (on the fact that this transition is the one that causes a state change) or unconditional. The conditional distributions would be used for a "competing process" model, where the state transitions represent independent processes, the first of which to finish determines which state transition occurs. The unconditional distributions would be used in a situation where the modeler knows the holding time distribution for each state and the transition probabilities for the state transitions. See section 5.2.10 for an example.

A semi-Markov chain is solved for the same measures as an acyclic Markov chain except that transient results are not available for irreducible semi-Markov chains. Furthermore, reward rates may be attached to the states of a semi-Markov model and corresponding reward-based measures can be computed.

Semi-Markov chains are used in section 5.2.11, 6.5 and 6.10.

2.8 Generalized Stochastic Petri Nets

A Petri Net consists of "places" containing "tokens" and transitions that specify how tokens move from one place to another (possibly multiplying themselves or being removed from the net in the process). A Generalized Stochastic Petri Net (GSPN) is Petri Net that allows the transitions to be either "immediate" (transition occurs as soon as enough tokens are present in the right places) or "timed". A timed transition has a "firing-time" distribution that might be non-exponential.

A GSPN is specified by listing

- place names along with the initial number of tokens in each place,
- the names of timed transitions along with the transition's distribution
- the names of immediate transitions
- arcs from places to transitions, along with the number of tokens that must be in the place for the transition to fire
- arcs from transitions to places, along with the number of tokens that appear in the the place when the transition fires

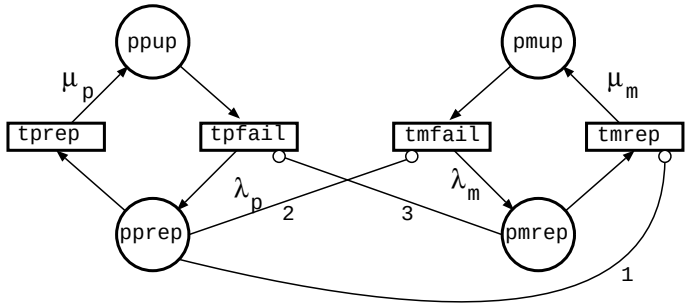


Figure 2.21: GSPN Availability Model

- the names of inhibitor arcs from places to transitions (tokens present in the place prevent the transition from firing), along with the number of tokens that must be in the place for the transition to be inhibited

GSPNs can be acyclic, irreducible and phase-type in a way analogous to Markov chains. SHARPE can compute (where the GSPN type is appropriate for the measure) steady-state average throughput and utilization for a transition, steady-state average number of tokens in a place, steady-state expected probability that a place is empty, and transient measures for expected number of tokens in a place, probability that a place is empty and throughput or utilization of a transition.

Continuing with the running example, we model availability using GSPN.Modeling the availability of this system with a GSPN does more than just give us another validity check. It would allow us to find the unavailability for a system with any number of processors and memories without having to construct a separate model for each number of components. The GSPN in Figure 2.21 is a model of the system in which there is one repair facility to be shared for all components.

There is a token for each processor and each memory. When a processor fails, its token moves from state *ppup* (place: processor up) through transition *tpfail* (transition: processor fails) to state *pprep* (place: processor waiting for repair). Processor repair is represented by a token moving from state *pprep* through transition *tprep* to state *ppup*. The inhibitor arcs from *pprep* to *tmfail* and *pmrep* to *tpfail* reflect the assumption that if the system has already failed because all processors or all memories have failed, the remaining working components do not fail while they aren't running. This aspect of the system was modeled only implicitly in the Markov chain model, by the absence of failure transitions from the states with either no operating processors or no operating memory modules. The inhibitor arc from *pprep* to *tmrep* is the one that represents our assumption that there is only one repair facility; if there are any failed processors, there can be no memory repair.

A SHARPE input file for this GSPN is shown in Figure 2.22. On line 1, we ask to have results shown to 8 decimal places (this is a formatting command only; it does not affect the precision with which calculations are made). On lines 10 through 41, we specify the GSPN in six sections. The first section (lines 12 through 15) specifies place names and the initial number of tokens in each place. The section section (lines 18 through 21) specifies timed transitions and their firing characteristics. Line 18 says that transition *tpfail* has a firing rate that is dependent (**dep**) on the number of tokens in place *ppup* and has parameter *lambdap*. This means that the distribution function for the time it takes transition *tpfail* to fire is exponentially distributed with parameter *k*lambdap* when there are *k* tokens in place *ppup*. Line 20 says that transition *tprep* has a firing rate independent (**ind**) of the number of tokens in any place, and the rate is *mup*. The third section specifies immediate transitions; there are none in this GSPN. The fourth section (lines 26 through 29) specifies input arcs (from places to transitions) and the number of tokens needed in the place to fire the transition. The fifth section (lines 32 through 35) specifies output arcs (from transitions to places) and the number of tokens that appear in the place when the transition fires. The sixth section (lines 38 through 40) specifies inhibitor arcs (places to transitions) and the number of tokens needed in the place to inhibit the transition.

On lines 43 through 46, we define our own function *ssunavailg* to give us the steady-state unavailability of the system. We make use of the SHARPE builtin function **prempty**, which gives the steady-state probability that a place contains no tokens. To illustrate the fact that it is easy to vary the number of processors or memories

```
 1  format 8
 2
 3  bind
 4  lambdap 1/720
 5  lambdam 1/(2*720)
 6  mup 1/4
 7  mum 1/2
 8  end
 9
10  gspn repg(nproc, nmem)
11  * places
12  ppup nproc
13  pmup nmem
14  pprep 0
15  pmrep 0
16  end
17  * timed transitions
18  tpfail dep ppup lambdap
19  tmfail dep pmup lambdam
20  tprep ind mup
21  tmrep ind mum
22  end
23  * immediate transitions
24  end
25  * input arcs
26  ppup tpfail 1
27  pmup tmfail 1
28  pprep tprep 1
29  pmrep tmrep 1
30  end
31  * output arcs
32  tpfail pprep 1
33  tmfail pmrep 1
34  tprep ppup 1
35  tmrep pmup 1
36  end
37  * inhibitor arcs
38  pprep tmrep 1
39  pprep tmfail nproc
40  pmrep tpfail nmem
41  end
42
43  func ssunavailg(p,m) \
44     prempty(repg,ppup;p,m) +\
45     prempty(repg,pmup;p,m) -\
46     prempty(repg,ppup;p,m)*prempty(repg,pmup;p,m)
47
48  loop p,2,10,2
49      expr ssunavailg(p,3)
50  end
51
52  end
```

Figure 2.22: SHARPE Input for GSPN

with the GSPN model, on lines 48 through 50 we use a loop to vary the number of processors from 2 to 10 and look at the steady-state unavailability. The results are as follows:

```
p=2.000000
     ssunavailg(p,3):   6.10665306e-05

p=4.000000
     ssunavailg(p,3):   4.75570390e-08

p=6.000000
     ssunavailg(p,3):   3.06582157e-08

p=8.000000
     ssunavailg(p,3):   3.67183540e-08

p=10.000000
     ssunavailg(p,3):   4.35124851e-08
```

We verified that analyzing this GSPN with two processors gave the same result for system steady-state unavailability as the Markov chain model. It took 0.2 processing-seconds on a NextStation (containing a 68040 processor) to get these results, and almost all of that time was spend on file input/output. We note that the GSPN, although a more efficient specification, is no more efficient to analyze than the Markov chain, since the SHARPE analysis of a GSPN involves translating the GSPN into a Markov chain.

For additional examples of the GSPN model, see section 5.2.5.

Chapter 3

Distribution Functions

One of SHARPE's most important features is that it produces model results as functions of one variable, usually time but sometimes cumulative reward. In this section, we explain the assumptions needed to allow this and why it works.

Suppose we have a system containing two independent components such that the system remains operational if at least one of the components has not failed. This could be modeled as a fault tree consisting of one AND gate with two inputs. If we knew that the likelihood that the components would fail within 6 months was $p_1 = .02$ for the first component and $p_2 = .04$ for the second, then the failure probability for the system during that time is $p_1 \times p_2 = .0008$.

Now suppose we knew the failure behavior in more detail. We might have access to failure data and a program for fitting a function to the data [35]. Suppose, for the sake of discussion, that we have fitted the failure data to two exponential distributions, $F_1(t) = 1 - e^{-0.004t}$ and $F_2(t) = 1 - e^{-0.005t}$, where $F_i(t)$ is the probability that component i has failed by time t. We can compute the system failure probability as a function of t by multiplying the two functions together, just as we multiplied the two probabilities together. The result is $F(t) = 1 - e^{-0.004t} - e^{-0.005t} + e^{0.009t}$.

The result is not another exponential distribution, but it does have the same form in that it is a sum of terms that look like ae^{bt} (where b might be 0). The algorithms for analyzing all of the models supported by SHARPE involve manipulating the probability distributions attached to components using addition, subtraction, multiplication, integration, differentiation and convolution (see section 2.12). With the exception of convolution, each of these operations when applied to terms of the form ae^{bt} results in more terms of the same form. If we take the convolution of two terms, we may get a term of the form ate^{bt}. Further application of the operations to these terms produce only terms of the form $at^k e^{bt}$. Functions consisting of sums of terms of this form are called "exponential polynomials", and they are closed under all of the operations the SHARPE algorithms need to use.

Because of this closure property, we can allow model components to be assigned probability distributions that are any exponential polynomial, and the model analysis yields another exponential polynomial. SHARPE allows the functions to be specified using any number of variable names, with the variable t being implicit. SHARPE computes symbolically only in t; it evaluates all other variables before beginning model analysis and produces a function symbolic in t. At a user's request, SHARPE will evaluate this function for any desired mission time t. To compute the mean or variance, SHARPE does the necessary integral computation symbolically rather than by using a numerical solution. Hierarchical modeling is supported by allowing users to assign a function produced by analyzing one model as the distribution of a basic component of another model.

Of course, not every exponential polynomial is a valid distribution function. There is no way to ensure that a given user-specified exponential polynomial is a valid probability distribution, but SHARPE does check that $0 \leq F(0) \leq lim_{t\to\infty} F(t) \leq 1$. We deliberately allow the possibility that $0 < F(0)$ and $lim_{t\to\infty} F(t) < 1$. A distribution with either or both of these conditions that is valid in every other respect is called a "defective" distribution. In later sections, we will give examples of where these arise and how they are handled. A single-point probability $F(t) = a, t \geq 0$ is a special, degenerate case of an exponential polynomial and is allowed. The a_i and b_i are allowed to be complex numbers, as long as the function is real-valued (the complex numbers must appear as conjugate pairs). In fact, this generality is needed if we are to analyze non-acyclic Markov chains for a solution symbol in t (see section ??).

In addition to the exponential distribution, the class of distribution functions with exponential polynomial form includes the hypoexponential, hyperexponential, Erlang and mixtures of Erlang distributions. In [43], it was shown that the class is identical to the Coxian distributions and is a proper superset of the class PH of phase-type distributions defined by Neuts [38].

For more information about exponential polynomials, see Appendix B.

Chapter 4

Reliability and Availability Analysis

Three common non-state-space model types used for reliability and availability analysis are series-parallel reliability block diagrams, fault trees and reliability graphs. Common state-space model types used for this purpose include Markov chains, semi-Markov chains and GSPN models. In this chapter, we present some examples of these model types and show how they are specified for SHARPE.

4.1 A Reliability Block Diagram

Consider a 5-stage system modeled by the reliability block diagram shown in Figure 4.1. Assume that the components behave in a statistically independent fashion. Figure 4.2(a) shows a SHARPE batch input file containing two specifications for this model. Remember that the line numbers are not (and cannot be) part of the files; they are included only so that we may easily identify lines while explaining the files.

In the specification called *example1a*, each component is assigned a probability value. This might be the probability that the component fails before some specific time t or that it fails before some particular mission is accomplished. Lines 1 through 10 define the structure of the reliability block diagram and assign probabilities to the component types. Line 1 identifies the type of model (**block**) and gives the model a name (*example1a*). Every model must have a name consisting of any printable characters except comma, semicolon and backslash. The names may be any length, but must be unique in the first 14 characters.

Lines 2 through 6 specify the five distinct component types in the block diagram. Each component type is identified with a name and assigned the probability of its failure by using the built-in distribution **prob**. Lines 7, 8 and 9 specify the structure of the block diagram. Line 10 marks the end of the block diagram specification using the keyword **end**. If this model had been entered interactively, line 10 would have been entered as a null line.

On line 12 of the input file, we ask to have an expression's value printed by using the keyword **expr**. An expression can involve variables, operators, parentheses, user-defined functions, and various built-in functions that SHARPE provides. In this case, we have the built-in function **sysprob**, which takes as its argument the name of a model whose components were assigned probabilities using the **prob** function. The interpretation of the value of **sysprob**(*example1a*) depends on the meaning of the probabilities attached to the component types. If each component type was assigned the probability that it has filed by a particular time, then **sysprob** (*example1a*) is the probability that the system has failed by that time.

Suppose that instead of assigning each component type a probability, we want to attach a time-to-failure

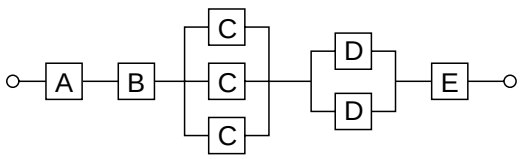
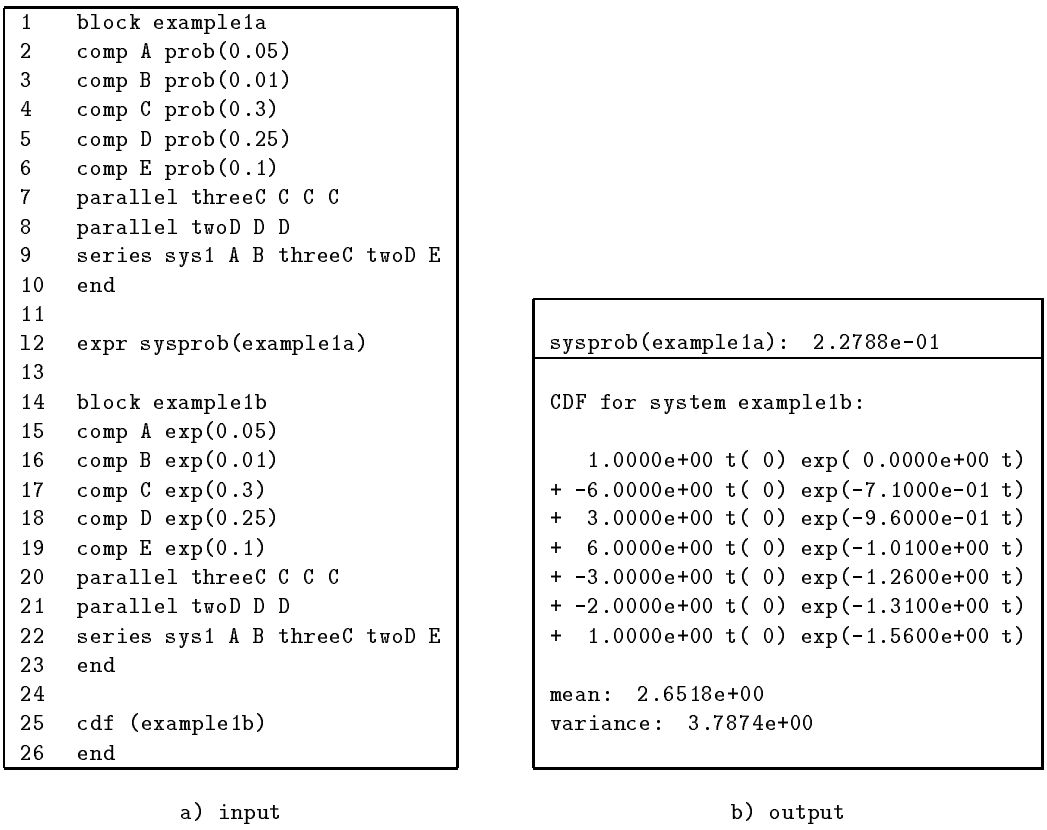


Figure 4.1: A Reliability Block Diagram



```
1   block example1a
2   comp A prob(0.05)
3   comp B prob(0.01)
4   comp C prob(0.3)
5   comp D prob(0.25)
6   comp E prob(0.1)
7   parallel threeC C C C
8   parallel twoD D D
9   series sys1 A B threeC twoD E
10  end
11
12  expr sysprob(example1a)
13
14  block example1b
15  comp A exp(0.05)
16  comp B exp(0.01)
17  comp C exp(0.3)
18  comp D exp(0.25)
19  comp E exp(0.1)
20  parallel threeC C C C
21  parallel twoD D D
22  series sys1 A B threeC twoD E
23  end
24
25  cdf (example1b)
26  end
```

a) input

```
sysprob(example1a):  2.2788e-01

CDF for system example1b:

   1.0000e+00 t( 0) exp( 0.0000e+00 t)
+ -6.0000e+00 t( 0) exp(-7.1000e-01 t)
+  3.0000e+00 t( 0) exp(-9.6000e-01 t)
+  6.0000e+00 t( 0) exp(-1.0100e+00 t)
+ -3.0000e+00 t( 0) exp(-1.2600e+00 t)
+ -2.0000e+00 t( 0) exp(-1.3100e+00 t)
+  1.0000e+00 t( 0) exp(-1.5600e+00 t)

mean:  2.6518e+00
variance:  3.7874e+00
```

b) output

Figure 4.2: Input and Output for Block Diagram

distribution to each one. For simplicity, we assume each distribution to be exponential. SHARPE provides the keyword **exp** to be used when assigning the exponential distribution. It has one parameter, which must be positive. The parameter is the inverse of the mean; in the case of block diagram components, it is the (constant) failure rate of the component.

Lines 14 through 23 show a SHARPE definition (*example1b*) that assigns exponential time-to-failure distributions to components. It is all right to use the same component type names that we used for *example1a*; the scope of a component type name is limited to the model in which it is defined.

On line 25, we ask to see the cumulative distribution function (keyword **cdf**) for the system. A CDF is computed in closed form as a function of t.

Figure 4.2(b) shows the output produced by SHARPE. First we see the system failure probability for the first model (*example1a*), then the failure distribution for the second model (*example1b*). The distibution is an exponential polynomial with seven terms:

$$1t^0e^{0t} - 6t^0e^{-0.71t} + 3t^0e^{-0.96t} + 6t^0e^{-1.01t} - 3t^0e^{-1.26t} - 2t^0e^{-1.31t} + 1t^0e^{-1.56t}.$$

Each term of the polynomial is printed on a separate line. The mean and variance of the time-to-failure of the system are also printed.

4.2 Emulating Relcomp

To illustrate the way in which a user defines distribution types, we will show how SHARPE can be made to look like the Relcomp (Reliability Computation) program [14], a simple block diagram analyzer. With the exception that SHARPE cannot handle the Weibull distribution, we can use SHARPE to analyze any Relcomp-type system,

and we can make appropriate definitions so that a user can specify the structure of a system in a way very similar to the Relcomp interface.

$R(t) = e^{-\lambda t}$	single exponential, with failure rate λ
$R(t) = 2e^{-\lambda t} - e^{-2\lambda t}$	*activeE* redundancy, 2 units with equal failure rates λ
$R(t) = e^{-\lambda_a t} + e^{-\lambda_b t} - e^{-(\lambda_a+\lambda_b)t}$	*activeU* redundancy, 2 units with unequal failure rates λ_a (primary) and λ_b (secondary)
$R(t) = e^{-\lambda t} + e^{-\lambda_s t}\lambda t e^{-\lambda t}$	*standbyE* redundancy, 2 units with equal failure rates λ and a sensing switch with failure rate λ_s
$R(t) = e^{-\lambda_a t} + e^{-\lambda_s t}\left(\frac{\lambda_a}{\lambda_b-\lambda_a}\right)\left(e^{-\lambda_a t} - e^{-\lambda_b t}\right)$	*standbyU* redundancy, 2 units with unequal failure rates λ_a (primary) and λ_b (secondary) and a sensing switch with failure rate λ_s
$R(t) = \sum_{i=0}^{m}\binom{n}{i}(e^{-\lambda t})^{n-i}(1-e^{-\lambda t})^i$	*binomial* (m-out-of-n system), with equal failure rates λ
$R(t) = p$	single, *oneshot* system, with probability p of success

Figure 4.3: Relcomp Reliability Functions

First, we describe the Relcomp program. At its top level, Relcomp computes the probability of survival (until a user-specified time t) for a series system in which there is no repair. Redundancy in the form of active or standby spare units is brought in by allowing the basic components in the series system to have reliability functions chosen from a set of pre-computed functions that were obtained by analyzing systems with redundancy. The set of reliability functions for components is shown in Figure 4.3. An example of a Relcomp-type system (taken from [14]) is shown in Figure 4.4.

The system of Figure 4.4 has four components in series; three of the components are internally redundant. This system would be specified for Relcomp by giving four component types: single exponential, active redundant equal, standby redundant equal, and binomial. The Relcomp program computes that the probability that the system is functioning at time $t = 20$ is 0.99581.

When using SHARPE to emulate Relcomp, it is important to recall that SHARPE computes a failure-time distribution rather than a reliability function. The Relcomp program computes the probability that the system is still operating at a specified time t. SHARPE computes the distribution function of the time-to-failure of the system; we can then evaluate the distribution at time t and subtract this value from one to obtain the reliability at time t.

The file in Figure 4.5 is a SHARPE input file that defines the distribution functions that are built into Relcomp.

Line 1 contains a comment; any line beginning with the character "*" is ignored. To define a distribution function, we use the keyword **poly** (for exponential polynomial), give the function a name and a parameter list, and then specify the the distribution function. Lines 3 through 6 define the distribution function *activeE*, having one parameter *lambda*. The keyword **gen** (for general exponential polynomial) tells SHARPE that we want

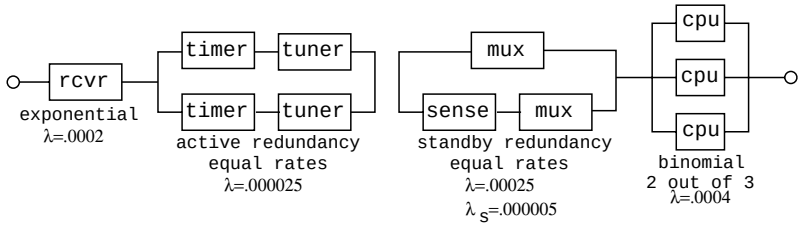


Figure 4.4: A Relcomp-type System

```
 1  * emulation of RELCOMP
 2
 3  poly activeE (lambda) gen \
 4    1,0,0 \
 5     -2,0,-lambda \
 6    1,0,-2*lambda
 7
 8  poly activeU (lambda,mu) gen \
 9    1,0,0 \
10   -1,0,-lambda \
11   -1,0,-mu \
12    1,0,-(lambda+mu)
13
14  poly standbyE (lambda, s) gen \
15    1,0,0 \
16   -1,0,-lambda \
17   -lambda,1,-(lambda+s)
18
19  poly standbyU (lambda,mu,s) gen \
20    1,0,0 \
21   -1,0,-lambda \
22    -lambda/(mu-lambda), 0, -(lambda+s) \
23    lambda/(mu-lambda), 0, -(mu+s)
24
25  poly oneshot(p) prob(1-p)
26
27  block KN (lambda, k, n)
28  comp x exp(lambda)
29  kofn top n-k+1,n, x
30  end
31
32  poly binomial (lambda, k, n) \
33     cdf (KN;lambda,k,n)
34
```

Figure 4.5: File Defining Relcomp Functions

to define the distribution giving all of the exponential polynomial terms in the sum $\sum_{j=1}^{n} a_j t^{k_j} e^{b_j t}$. SHARPE requires the definition of a distribution function to be all on one line, but as was mentioned previously it allows the UNIX-style use of the backslash character for line continuation. Each of lines 4 through 6 contains one term, in the order a_j, k_j, b_j. The three terms of the exponential polynomial *activeE* specify the distribution

$$1t^0e^{0t} - 2t^0e^{-\lambda t} + 1t^0e^{-2\lambda t} = 1 - [2e^{-\lambda t} - e^{-2\lambda t}].$$

Lines 8 through 23 similarly define the distribution functions for *activeU*, *standbyE* and *standbyU* systems.

The fifth definition is for the component type *oneshot*, which is assigned a probability value rather than a distribution function. This definition uses the built-in distribution type **prob**, having one parameter. The parameter of the distribution **prob** gives the probability that the component fails immediately; if it does not fail immediately, then it never fails. (For more information about this distribution, see Appendix B.)

To provide the binomial distribution we must do more work (lines 27 through 33). First, we define a block diagram system called *KN*. The system has one basic component x having exponential distribution with parameter *lambda*. The binomial distribution results when n identically distributed independent copies of x are combined in such a way that the system of n components fails whenever k or more of the components have failed. That means that the system is operational whenever $n - k + 1$ or more components are operational. This structure is expressed on line 29. Once the system *KN* is defined, we can use the system to define the distribution called *binomial* (line 32); it is defined to have the distribution of the system *KN* when evaluated with parameters *lambda, k,* and *n*.

The file in Figure 4.6(a) specifies the model shown in Figure 4.4 and asks for the probability of it being operational at time $t = 20$. The Relcomp program obtains the probability by analyzing the system numerically, using numerical integration (Simpson's rule) to compute the answer for the particular value of t. To compute the reliability for a different value of t, Relcomp would have to do the entire computation over again. When we use SHARPE to analyze the system, SHARPE computes (symbolically in t) the complete distribution function of the failure time for the system, and then evaluates the function at the particular time desired. With no additional work, SHARPE can print the entire distribution function.

Figure 4.6(b) shows what the output looks like. The reliability at time t=20 matches that obtained by the Relcomp program.

We could use the power of SHARPE to provide extensions to the Relcomp model. For example, we could allow k-out-of-n systems where the components were not identically distributed. We could also, by using Markov submodels, allow redundant systems with varying numbers of active and standby spares.

```
 1  include relcomp
 2
 3  block DP
 4
 5  comp receiver exp(.0002)
 6  comp tuner activeE(.000025)
 7  comp mux standbyE(.00025, .000005)
 8  comp cpu binomial(.0004, 2, 3)
 9  series DP receiver tuner mux cpu
10  end
11
12  expr 1-value(20;DP)
13  cdf(DP)
14  end
```

a) input

```
1-value(20;DP): 9.9581e-01

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


CDF for system DP:

   -1.5000e-03 t( 1) exp(-1.2800e-03 t)
 +  7.5000e-04 t( 1) exp(-1.3050e-03 t)
 +  1.0000e-03 t( 1) exp(-1.6800e-03 t)
 + -5.0000e-04 t( 1) exp(-1.7050e-03 t)
 +  1.0000e+00 t( 0) exp( 0.0000e+00 t)
 + -6.0000e+00 t( 0) exp(-1.2750e-03 t)
 +  3.0000e+00 t( 0) exp(-1.3000e-03 t)
 +  4.0000e+00 t( 0) exp(-1.6750e-03 t)
 + -2.0000e+00 t( 0) exp(-1.7000e-03 t)

mean:  1.4794e+03
variance:  1.2465e+06
```

b) output

Figure 4.6: Input and Output Files for Relcomp-type System

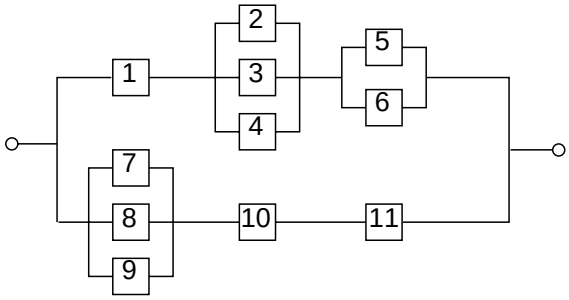


Figure 4.7: Another Reliability Block Diagram

4.3 Instantaneous Availability

If a system is composed of components each having an independent repair facility, SHARPE can be used to compute the instantaneous availability of the system. Consider the series-parallel system of components pictured in Figure 4.7

This is the example presented in [37], where an approximation method is given for computing the steady state unavailability of a series-parallel system. Using our model, we can compute the steady state unavailability exactly, and in addition we compute the transient unavailabilities.

Assume that each component is subject to failure, and has its own independent repair facility. If the time-to-failure of component i is exponentially distributed with failure rate λ_i and the time-to-repair is exponentially distributed with repair rate μ_i, then the instantaneous availability is [53]

$$A_i(t) = \frac{\mu_i}{\lambda_i + \mu_i} + \frac{\lambda_i}{\lambda_i + \mu_i} e^{-(\lambda_i + \mu_i)t}.$$

As t approaches infinity, $A_i(t)$ approaches the steady-state availability. If $\mu_i = 0$ (no repair), $A_i(t)$ reduces to the reliability (as a function of time) of the component.

```
bind                                    poly unavail(mu,lambda) gen \
lambda1 .001                                 1,0,0 \
lambda2 .01                                  -mu/(lambda+mu),0,0 \
lambda3 .01                                  -lambda/(lambda+mu),0,-(lambda+mu)
lambda4 .01
lambda5 .005                            block inst_avail
lambda6 .005                            comp 1 unavail(mu1,lambda1)
lambda7 .01                             comp 2 unavail(mu2,lambda2)
lambda8 .01                             comp 3 unavail(mu3,lambda3)
lambda9 .01                             comp 4 unavail(mu4,lambda4)
lambda10 .01                            comp 5 unavail(mu5,lambda5)
lambda11 .01                            comp 6 unavail(mu6,lambda6)
mu1 1/5                                 comp 7 unavail(mu7,lambda7)
mu2 1/7.5                               comp 8 unavail(mu8,lambda8)
mu3 1/7.5                               comp 9 unavail(mu9,lambda9)
mu4 1/7.5                               comp 10 unavail(mu10,lambda10)
mu5 1/6                                 comp 11 unavail(mu11,lambda11)
mu6 1/6
mu7 1/7.5                               parallel 56 5 6
mu8 1/7.5                               parallel 234 2 3 4
mu9 1/7.5                               parallel 789 7 8 9
mu10 1/5                                series 1-6 1 234 56
mu11 1/5                                series 7-11 789 10 11
end                                     parallel all 1-6 7-11
                                        end

                                        cdf(inst_avail)
```

Figure 4.8: Input for the Instantaneous Unavailability Example

Let the distribution function associated with component i be $U_i(t) = 1 - A_i(t)$. This distribution represents the instantaneous unavailability of the component and is in exponential polynomial form with a mass at infinity. We want to compute the instantaneous unavailability of the system as a whole. For subsystems in parallel, we must take the product of the component unavailabilities (the system is unavailable only when all parallel subsystems are unavailable). This is the "maximum" combination. For a series of components, the availability is the product of the component availabilities (the system is available only when all subsystems are available). Thus, the unavailability of the system is exactly the "minimum" combination of the components.

Because the combining operations are exactly "maximum" and "minimum", we can use SHARPE to compute $U(t)$, the instantaneous unavailability for the system as a whole. By taking the limit of $U(t)$ as t approaches infinity, we obtain the steady-state system unavailability, and by setting all $\mu_i = 0$, we obtain system unreliability as a function of the mission time t. Note that μ_i may be zero for some or all of the components, and we still obtain the instantaneous unavailability for the overall system.

We used SHARPE to compute the unavailability for this system using the same parameters as in [37]. A SHARPE input file for this example is shown in Figure 4.8. The parameters are $\lambda_5 = \lambda_6 = .005$, $\lambda_1 = .001$, $\lambda_i = .01$ for all other i, $\mu_5 = \mu_6 = 1/6$, $\mu_1 = \mu_{10} = \mu_{11} = 1/5$, and $\mu_i = 1/7.5$ for all other i. The steady state unavailability is computed to be 5.743010^{-4}. It should be noted that the approximation given in [37] is in error. The approximation as computed using the method in [37] should be 7.102110^{-4}.

Because SHARPE distributions can contain complex numbers, we can allow the failure or repair time distributions to be non-exponential. Suppose the failure time distribution for a component i is 2-stage Erlang with parameter $2\lambda_i$ and the repair time distribution is exponential with rate μ_i. Then its instantaneous availability is given by:

$$A_i(t) = \frac{\mu_i}{\lambda_i + \mu_i} + \frac{\lambda_i}{\lambda_i + \mu_i} \left(\frac{\theta_{i1} - \lambda_i - \mu_i}{\theta_{i1} - \theta_{i2}} e^{-\theta_{i1} t} + \frac{\theta_{i2} - \lambda_i - \mu_i}{\theta_{i2} - \theta_{i1}} e^{-\theta_{i2} t} \right)$$

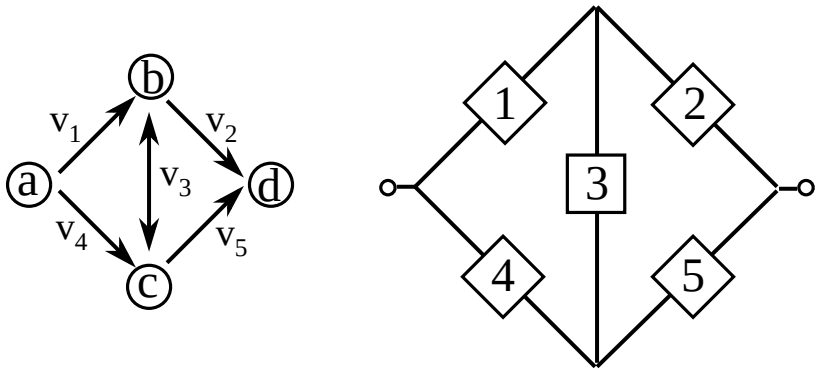


Figure 4.9: A Reliability Graph and Equivalent Non-series-parallel Block Diagram

where

$$\theta_{i1}, \theta_{i2} = \frac{4\lambda_i + \mu_i \pm \sqrt{\mu_i(\mu_i - 8\lambda_i)}}{2}.$$

If $\mu_i < 8\lambda_i$, the above function will contain complex numbers, a situation that SHARPE allows.

4.4 A Reliability Graph

A reliability graph consists of nodes and directed arcs. One node is the "source", meaning no arcs enter it, and one node is the "sink", meaning no arcs leave it. A system represented by a reliability graph fails when there is no path from the source to the sink. The arcs are assigned failure distributions.

The left-hand side of Figure 4.9 shows a reliability graph model. The system it represents is operational if there is a path from *a* to *d*; the possible paths are *abd*, *acd*, *abcd* and *acbd*. The system fails if the arcs labeled *v1* and *v4* both fail or if the arcs *v2* and *v5* both fail.

This reliability graph is equivalent to the non-series-parallel reliability block diagram on the right-hand side of Figure 4.9; this particular block diagram is called a "bridge". In the reliability graph, the components are the arcs, while in the block diagram the components are the boxes. The block diagram cannot be analyzed by (or even specified for) SHARPE, but the reliability graph can.

Suppose the arcs in the reliability graph are assigned exponential time-to-failure distributions with parameters v_i. Figure 4.10(a) shows a SHARPE input file for the reliability graph.

On lines 3 through 6, we specify four unidirectional arcs by giving, on each line, an ordered pair of nodes followed by a distribution function. The keyword **bidirect** on line 7 means that all succeeding lines define bidirectional arcs. If a reliability graph contains only unidirectional arcs, this keyword need not be present. On line 11 we ask for the time-to-failure distribution for the system as an exponential polynomial. On line 12, we use the keyword **pqcdf** to ask for the failure distribution in terms of the failure distributions of the individual arcs. The name **pqcdf** comes from the fact that it is a "CDF" (Cumulative Distribution Function) in terms of probabilities of failure (often denoted "p") and non-failure (often denoted "q" where q=1-p) of the individual components. The algorithms SHARPE uses for analyzing reliability graphs and fault trees with repeated nodes can produce a result in the "pq" format; the format is not available for other model types.

Results are shown in Figure 4.10(b). In the "pq" form of the distribution, **P(0:a,b)** is the probability that the arc from **a** to **b**, which SHARPE has internally numbered **0**, is functioning. On the fourth line of the form, **Q(3:c,d)** is the probability that the arc from **c** to **d**, internally numbered **3**, is not functioning.

4.5 A Fault Tree

We consider a reliability model proposed in [7]. The system being modeled consists of two components, A and B, combined in series. Assume that the distribution of the time-to-failure of component A is hyperexponential:

```
1    relgraph bridge(v1,v2,v3,v4,f5)
2
3    a b exp(v1)
4    b d exp(v2)
5    a c exp(v4)
6    c d exp(v5)
7    bidirect
8    b c exp(v3)
9    end
10
11   cdf (bridge;1,2,3,4,5)
12   pqcdf (bridge;1,2,3,4,5)
13
14   end
```

a) input

```
CDF for system bridge:

   1.0000e+00 t( 0) exp( 0.0000e+00 t)
+ -1.0000e+00 t( 0) exp(-3.0000e+00 t)
+ -3.0000e+00 t( 0) exp(-9.0000e+00 t)
+  1.0000e+00 t( 0) exp(-1.0000e+01 t)
+  1.0000e+00 t( 0) exp(-1.1000e+01 t)
+  1.0000e+00 t( 0) exp(-1.2000e+01 t)
+  1.0000e+00 t( 0) exp(-1.3000e+01 t)
+  1.0000e+00 t( 0) exp(-1.4000e+01 t)
+ -2.0000e+00 t( 0) exp(-1.5000e+01 t)

mean:  3.7741e-01 variance:  9.9183e-02
-----------------------------------------

CDF for system bridge:

1 − ([P(0 : a, b) ∗ P(1 : b, d)]+
[P(2 : a, c) ∗ P(3 : c, d) ∗ (1 − P(0 : a, b) ∗ P(1 : b, d))]+
[P(0 : a, b) ∗ Q(1 : b, d) ∗ Q(2 : a, c) ∗ P(3 : c, d) ∗ P(4 : b, c)]+
[Q(0 : a, b) ∗ P(1 : b, d) ∗ P(2 : a, c) ∗ Q(3 : c, d) ∗ P(4 : b, c)])
```

b) output

Figure 4.10: Input and Output File for Bridge

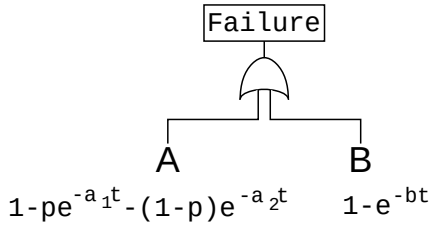


Figure 4.11: A Fault Tree for a Series System

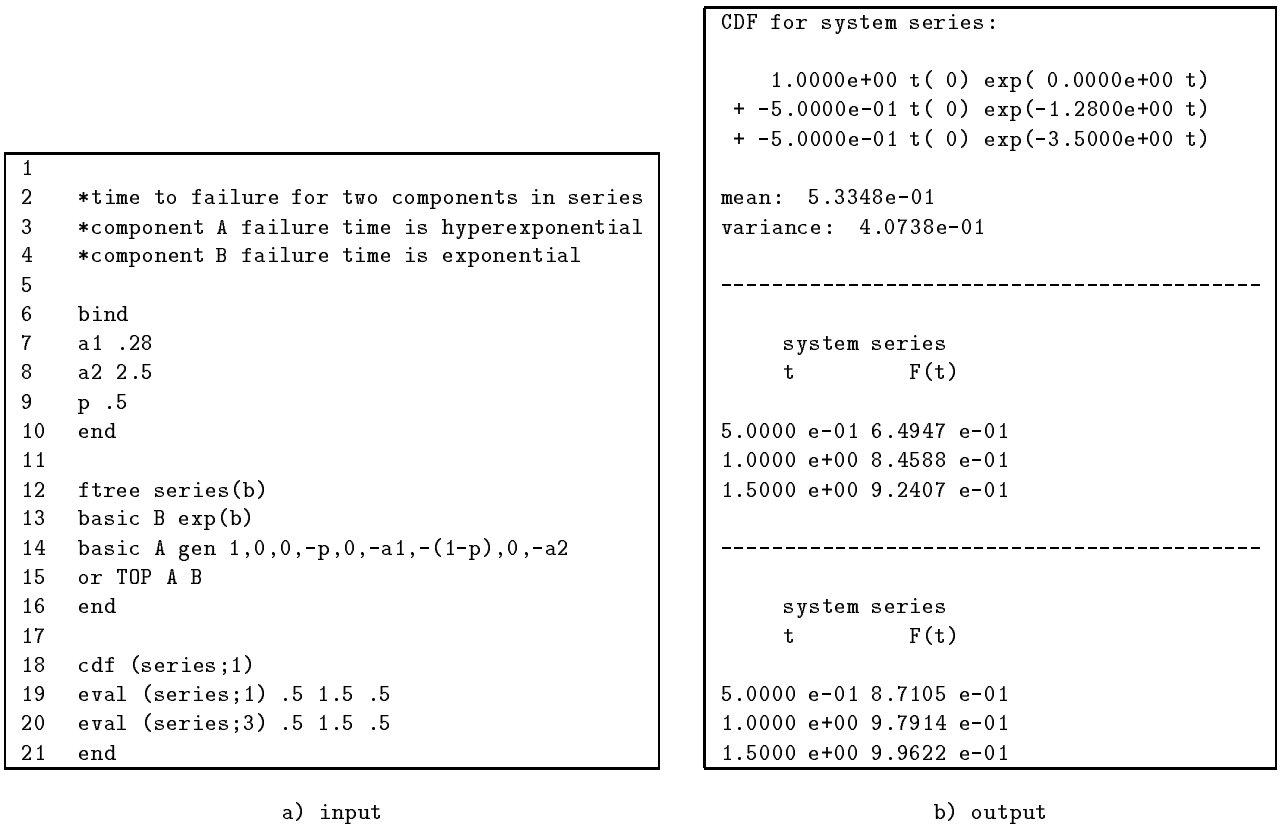
$F(t) = 1 - pe^{-a_1 t} - (1-p)e^{-a_2 t}$, and that of component B is exponential. A series system with two components A and B can be modeled using a very simple fault tree (Figure 4.11).

The tree has two basic components joined by an "or" gate; the system fails if either of the two components fails. The fault tree model can be written for SHARPE as shown in Figure 4.12(a).

On lines 6 through 10, variables are given values. On line 12, we define a fault tree system by using the keyword **ftree,** give the system a name, and define a parameter b for the system.

On line 13, we define the component B. A component is defined by the keyword **basic**, followed by the component name, and then by a distribution specification. Component B is assigned the exponential distribution. On line 14, we define the component A. To specify the hyperexponential distribution, the distribution type **gen** (general exponential polynomial) is used. This tells SHARPE that we want to define our own distribution. The distribution type **gen** allows us to specify an exponential polynomial where all of the parameters are real numbers. The input line must continue with a sequence of triples, each triple giving one of the terms a_j, k_j, b_j. Thus the distribution attached to event A is $1t^0e^{0t} - pt^0e^{-a_1 t} - (1-p)t^0e^{-a_2 t}$. On line 15, we define the "or" gate combining the two components. The gate is defined using the keyword **or**, given the name *TOP*, and has inputs A and B. Every gate must be given a name.

On line 18, we ask to see the CDF of the fault tree. For a fault tree, the CDF is that of the time-to-failure for the system. Since the defined system has a parameter, the name *series* is followed by a semicolon and then an



```
1
2    *time to failure for two components in series
3    *component A failure time is hyperexponential
4    *component B failure time is exponential
5
6    bind
7    a1 .28
8    a2 2.5
9    p .5
10   end
11
12   ftree series(b)
13   basic B exp(b)
14   basic A gen 1,0,0,-p,0,-a1,-(1-p),0,-a2
15   or TOP A B
16   end
17
18   cdf (series;1)
19   eval (series;1) .5 1.5 .5
20   eval (series;3) .5 1.5 .5
21   end
```

a) input

```
CDF for system series:

    1.0000e+00 t( 0) exp( 0.0000e+00 t)
 + -5.0000e-01 t( 0) exp(-1.2800e+00 t)
 + -5.0000e-01 t( 0) exp(-3.5000e+00 t)

mean:  5.3348e-01
variance:  4.0738e-01

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

     system series
     t         F(t)

5.0000 e-01 6.4947 e-01
1.0000 e+00 8.4588 e-01
1.5000 e+00 9.2407 e-01

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

     system series
     t         F(t)

5.0000 e-01 8.7105 e-01
1.0000 e+00 9.7914 e-01
1.5000 e+00 9.9622 e-01
```

b) output

Figure 4.12: Input and Output File for Fault Tree Example

argument. When the system *series* is analyzed, the parameter b takes on the value 1. Although in this example the argument is a constant value, in general, an argument can be any expression as long as all variables in it have been given values. On line 19, we ask to have the CDF evaluated over between 0.5 and 1.5 at intervals of 0.5 with b set to 1. SHARPE will see that the parameter has the same value as before and will not reanalyze the system. On line 20, we ask to have the system CDF evaluated when b has the value 3. SHARPE will recognize that the parameter has a different value and reanalyze the system to obtain a new CDF for evaluation. Figure 4.12(b) shows the results produced by SHARPE for this input file.

4.6 Another Fault Tree

Another example of a fault tree is the system specified in the input file in Figure 4.13(a). This system consists of *nsys* redundant subsystems configured in parallel, such that the system fails if *ksys* or more subsystems have failed. Each of these subsystems uses n-fold redundancy with the proviso that k or more must fail for the subsystem to have failed. The time-to-failure distributions for the components are independent and exponentially distributed. Note that for the top gate we have a *ksys/nsys* gate with dissimilar distributions attached to the gate inputs. The SHARPE command **format** controls the number of "significant" digits printed by SHARPE when it prints results. It has no affect on the precision of the internal calculations, which are carried out using whatever format the hardware and software uses to implement the C data type **double**. The output for this example is shown in Figure 4.13(b).

```
ftree fcc1
basic A exp(lambda)
basic B exp(mu)
basic C exp(sigma)
kofn Afail k,n, A
kofn Bfail k,n, B
kofn Cfail k,n, C
kofn Sysfail ksys,nsys, \
        Afail Bfail Cfail
end

bind
k 2
n 3
ksys 2
nsys 3
lambda 0.00001
mu 0.00002
sigma 0.00003
end

format 8

cdf (fcc1)
eval (fcc1) 10 60 10
end
```

a) input

```
CDF for system fcc1:

    1.00000000e+00 t( 0) exp( 0.00000000e+00 t)
 + -9.00000000e+00 t( 0) exp(-6.00000000e-05 t)
 +  6.00000000e+00 t( 0) exp(-7.00000000e-05 t)
 + -3.00000000e+00 t( 0) exp(-8.00000000e-05 t)
 +  2.00000000e+00 t( 0) exp(-9.00000000e-05 t)
 + -9.00000000e+00 t( 0) exp(-1.00000000e-04 t)
 +  6.00000000e+00 t( 0) exp(-1.10000000e-04 t)
 +  5.60000000e+01 t( 0) exp(-1.20000000e-04 t)
 + -3.00000000e+01 t( 0) exp(-1.30000000e-04 t)
 + -3.60000000e+01 t( 0) exp(-1.40000000e-04 t)
 + -1.60000000e+01 t( 0) exp(-1.50000000e-04 t)
 +  2.40000000e+01 t( 0) exp(-1.60000000e-04 t)
 +  2.40000000e+01 t( 0) exp(-1.70000000e-04 t)
 + -1.60000000e+01 t( 0) exp(-1.80000000e-04 t)

mean:  4.06425437e+04
variance:  4.71105707e+08

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

      system fcc1
        t             F(t)

 1.00000000 e+01 4.32986980 e-14
 2.00000000 e+01 7.05213665 e-13
 3.00000000 e+01 3.56403795 e-12
 4.00000000 e+01 1.12529985 e-11
 5.00000000 e+01 2.74560374 e-11
 6.00000000 e+01 5.68869396 e-11
```

b) output

Figure 4.13: Input and Output for Another Fault Tree

4.7 Aircraft Flight Control System

We consider a somewhat modified version of the flight control system modeled in Appendix G of [4]. The system contains three inertial reference sensors (IRS) and three pitch rate sensors (PRS), that monitor the status of the aircraft. All of the sensors are connected to each of four computer systems (CS). The computer systems independently collect information from the sensors and process the information. The computers are connected to each other and to three secondary actuators (SA). At least two of each type of component must be operational in order for the overall system to function correctly. In this system, the computers are most susceptible to failure (the failure rate for a computer is an order of magnitude greater than the failure rates for any of the other components). That is why there are four computers and only three of each of the other component types. A fault tree for this system is shown in Figure 4.14

Figure 4.15 shows input and output of SHARPE for this flight control system. The failure probability for a 10 hour mission is 1.02381×10^{-6}.

4.8 An Acyclic Markov Chain

To illustrate the use of acyclic Markov chains, we consider a Markov chain version of the fault tree presented in Section 4.5. The Markov model was proposed in [7]. The system being modeled consists of two components,

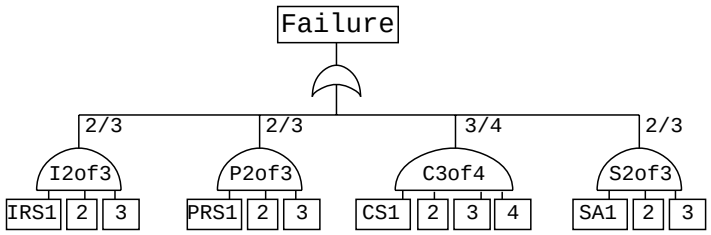


Figure 4.14: A Fault Tree for Aircraft Control System

```
* aircraft flight control system

bind
mIRS .000015
mPRS .000019
mSA .000037
mCS .00048
end

ftree aircraft
basic IRS exp(mIRS)
basic PRS exp(mPRS)
basic CS exp(mCS)
basic SA exp(mSA)
kofn IRS23 2,3, IRS
kofn PRS23 2,3, PRS
kofn CS34 3,4, CS
kofn SAS23 2,3, SA
or TOP IRS23 PRS23 CS34 SAS23
end

format 8

expr mean(aircraft)
eval (aircraft) 1000 10000 1000
expr value(10;aircraft) end
```

input

```
   mean(aircraft):    2.21322439e+03
-------------------------------------------

        system aircraft
          t             F(t)

   1.00000000 e+03 1.62926398 e-01
   2.00000000 e+03 5.15384469 e-01
   3.00000000 e+03 7.70884601 e-01
   4.00000000 e+03 9.02767485 e-01
   5.00000000 e+03 9.61222540 e-01
   6.00000000 e+03 9.85111991 e-01
   7.00000000 e+03 9.94421933 e-01
   8.00000000 e+03 9.97944094 e-01
   9.00000000 e+03 9.99250932 e-01
   1.00000000 e+04 9.99729373 e-01
-------------------------------------------

value(10;aircraft):  1.02381366e-06
```

output

Figure 4.15: Input and Output for Aircraft Control System

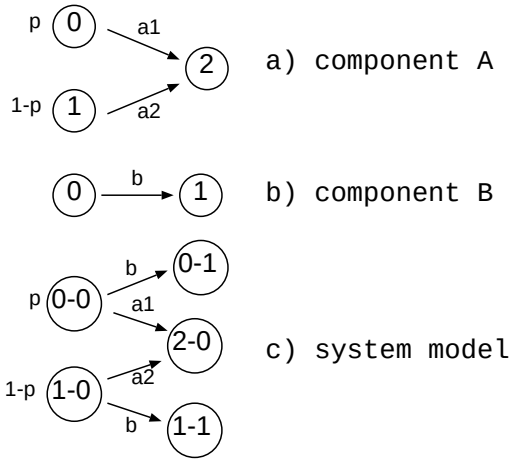


Figure 4.16: A Markov Model for a Series System

A and *B,* combined in series. Component *B* has two possible states: operating (state 0) and failed (state 1). Component *A* has three possible states: in states 0 and 1 it is operational and in state 2 it has failed. *A* starts operation in state 0 with probability *p* and in state 1 with probability 1-*p*. Figure 4.16 (a) and (b) show the failure processes of components *A* and *B,* respectively.

The Markov chain in Figure 4.16 (c) can be used to find the time-to-failure for the series system. Each state is an ordered pair, with the first element giving the condition of component *A* and the second giving the condition of component *B.* The system starts in state *0-0* with probability *p* and in state *1-0* with probability *1-p.* The states *2-0, 0-1,* and *1-1* are failure states.

Figure 4.17(a) shows an input file for this model. Lines 1 through 3 are comments. On lines 5 through 10, we assign values to some simple variables. Lines 12 through 21 define the Markov chain. Line 12 identifies the type of the model (**markov**) and gives the model a name. It is not necessary to specify the type of the Markov chain; SHARPE will determine that it is acyclic and analyze it appropriately.

Lines 13 through 16 give all the state transitions in the chain and their associated transition rates. Each line is a triple: *from-state, to-state, transition-rate.* Like model names, state names may contain any characters except comma, semicolon, and backslash. The lines in this section can appear in any order; the arcs do not have to be presorted. Line 17 marks the end of the arc/rate specification.

Once the arcs and rates are specified, we must specify the initial state probabilities for the Markov chain. These are defined in lines 19 through 21. The probabilities must add up to one. Note that the probability specification for state *1-0* is an expression rather than a simple variable. In fact, SHARPE allows the use of arbitrary expressions (using addition, subtraction, multiplication, division, exponentiation, and parentheses) any place where a scalar quantity (as opposed to a distribution function) is needed.

Line 23 requests that the CDF of the model *series* be printed. When SHARPE reads this line, it will analyze the system *series* and print the CDF of the time-to-absorption, along with the mean and the variance. Since this is a reliability model, the CDF is that of the time-to-failure of the series system. On line 24, we ask to have the CDF evaluated over the interval from 0.5 to 1.5, with increments of 0.5. SHARPE will not reanalyze the system; it simply evaluates the CDF at the requested values of *t*.

Suppose we are interested in the effect of a larger value of *b* on the reliability of the system. We can specify another value for *b* by rebinding it and requesting that the system CDF be evaluated for the new value of *b.* On lines 26 through 30 we rebind *b* and ask for a new evaluation of the CDF. SHARPE reanalyzes a system when an evaluation is requested and any of the variables are rebound.

Note that we have rebound *b* without rebinding the other variables. The other variables are assumed to retain their original values. Once a variable is bound to a value, it retains that value until (and unless) it is rebound to a different value.

Now suppose we are interested in the probability that a system failure is due to a failure in component *A*, and we would like to know how long it takes before the system fails in that case. We can get that information by asking SHARPE to print the CDF for the time to reach the state *2-0* in the Markov chain. This is done on

```
 1  * system reliability for A and B in series
 2  * failure time for B is exponential
 3  * failure time for A is hyperexponential
 4
 5  bind
 6  a1 .28
 7  a2 2.5
 8  b 1
 9  p .5
10  end
11
12  markov series
13  0-0 2-0 a1
14  1-0 2-0 a2
15  0-0 0-1 b
16  1-0 1-1 b
17  end
18
19  0-0 p
20  1-0 1-p
21  end
22
23  cdf(series)
24  eval(series) .5 1.5 .5
25
26  bind
27  b 3
28  end
29
30  eval(series) 0.5 1.5 0.5
31
32  cdf(series,2-0)
33  expr prob(series,0-1) + prob(series,1-1)
34  end
```

a) input

```
 1
 2  CDF for system series:
 3
 4  1.0000e+00 t( 0) exp( 0.0000e+00 t)
 5  + -5.0000e-01 t( 0) exp(-1.2800e+00 t)
 6  + -5.0000e-01 t( 0) exp(-3.5000e+00 t)
 7
 8  mean:  5.3348e-01
 9  variance:  4.0738e-01
10
11  ----------------------------------------
12
13  system series
14  t F(t)
15
16  5.0000 e-01 6.4947 e-01
17  1.0000 e+00 8.4588 e-01
18  1.5000 e+00 9.2407 e-01
19
20  ----------------------------------------
21
22  system series
23  t F(t)
24
25  5.0000 e-01 8.7105 e-01
26  1.0000 e+00 9.7914 e-01
27  1.5000 e+00 9.9622 e-01
28
29  ----------------------------------------
30
31  information about system series node 2-0
32
33  probability of entering node:  2.6996e-01
34
35  conditional CDF for time of reaching this state
36
37  1.0000e+00 t( 0) exp( 0.0000e+00 t)
38  + -1.5811e-01 t( 0) exp(-3.2800e+00 t)
39  + -8.4189e-01 t( 0) exp(-5.5000e+00 t)
40
41  mean:  2.0128e-01
42  variance:  4.4543e-02
43
44  ----------------------------------------
45
46  prob(series,0-1) + prob(series,1-1):  7.3004e-01
47
```

b) output

Figure 4.17: Input and Output for Acyclic Markov Chain

line 32. SHARPE does not have to do any additional computation to respond to this request.

On line 33, we ask for the value of an expression that give us the probability that a system failure is due to a failure in component *B*. The built-in function **prob** (not to be confused with the built-in distribution function with the same name) takes as its first argument the name of a Markov or semi-Markov system. The second argument must be the name of a state in that system. The value of **prob** is the probability that the given state is ever visited. Here we ask for the sum of the probability of visiting state *0-1* and the probability of visiting state *1-1.*

The input file ends with the keyword **end** on line 34, indicating there are no further definitions or requests.

Figure 4.17(b) shows the output produced by SHARPE. Lines 2 through 9 are the response to line 23 of the input file, giving the CDF for the system. SHARPE identifies the system *series* on line 2, gives the CDF of the time to absorption in the form of an exponential polynomial on lines 4 through 6, and gives the mean and variance of the CDF on lines 8 and 9.

The next two sets of results are for the two requests for evaluation of the system CDF (lines 24 and 30 of the input file). Each time, the system is identified by name and the results are given. Lines 31 through 42 are the response to line 32 of the input file, which requested the CDF for the state *2-0*. The system and state are identified on line 31. Line 33 gives the probability of reaching the state, and lines 37 through 42 give the conditional CDF (and its mean and variance) for the time to reach this state, given that this state is ever reached. The last result is the response to the request for the value of an expression (input file line number 33). SHARPE echoes the expression as it appeared in the input file and then gives the value of the expression. The probability that the system failed due to a failure in component *B* is 0.73004.

4.9 A Markov Chain with Function Definitions

For this example, we consider a model of the fault-tolerant SIFT (Software Implemented Fault Tolerance) system [60]. Suppose we have a computer system that contains 4 identical processors and 3 identical bus systems. The system can detect when a processor or bus has failed, and it can reconfigure so that the failed component is removed. There must be at least 2 processors and 2 buses working in order for the system to be operational.

Suppose reconfiguration fails if and only if, during reconfiguration, a second failure occurs of the same component type as the failed component. Then the coverage (probability that reconfiguration is successful) depends on the state of the system, since the more components there are in the system, the more likely it is that a second, near-coincident fault occurs while processing the first fault. If the component (processor or bus) failure rate is x, there are k operational copies of the component, and the reconfiguration rate is α, then the coverage is given by (see [54])

$$c(x,k) = \frac{\alpha}{\alpha + (k-1)x}.$$

Figure 4.18 shows a Markov chain that models this system.

For clarity, the failure state *F* has been shown twice. In state (*i,j*), *i* processors and *j* buses are operational. Figure 4.19 shows how this chain can be specified for SHARPE.

On line 3, we define the coverage function *c(x,k)* as defined above, with $\alpha = 360$. On line 4, we define the rate *r(x,k)* for a successful reconfiguration. On line 5, we define the rate *f(x,k)* for a failure due to unsuccessful reconfiguration. Note that the functions *r* and *f* are defined in terms of the function *c.* In this example, all three functions happen to have two parameters. SHARPE allows functions to be defined with any number of parameters, including the special case of no parameters.

On lines 7 through 26, we define the Markov chain. Note that this definition contains no initial state probabilities. This example is typical of many Markov chain models, in that there is only one "start" state (having no predecessors). In this case, the user is allowed to leave out the starting probabilities, and SHARPE will assume that the initial probability is 1 for the unique start state and 0 for all other states. Note that line 26, marking the end of the initial state probability specification, must be present even if no probabilities are given.

The built-in functions **mean** and **variance** can be used to get the mean and variance of the conditional distribution for the time to reach or leave (depending on whether the state is absorbing or not) a state in a Markov or semi-Markov chain. The name of the chain would be followed by a comma and the name of the state of interest. If the system had been declared with parameters, the system or state name would be followed by a semicolon and then an argument list. On lines 33 and 34, we ask for the mean and variance for the distribution of

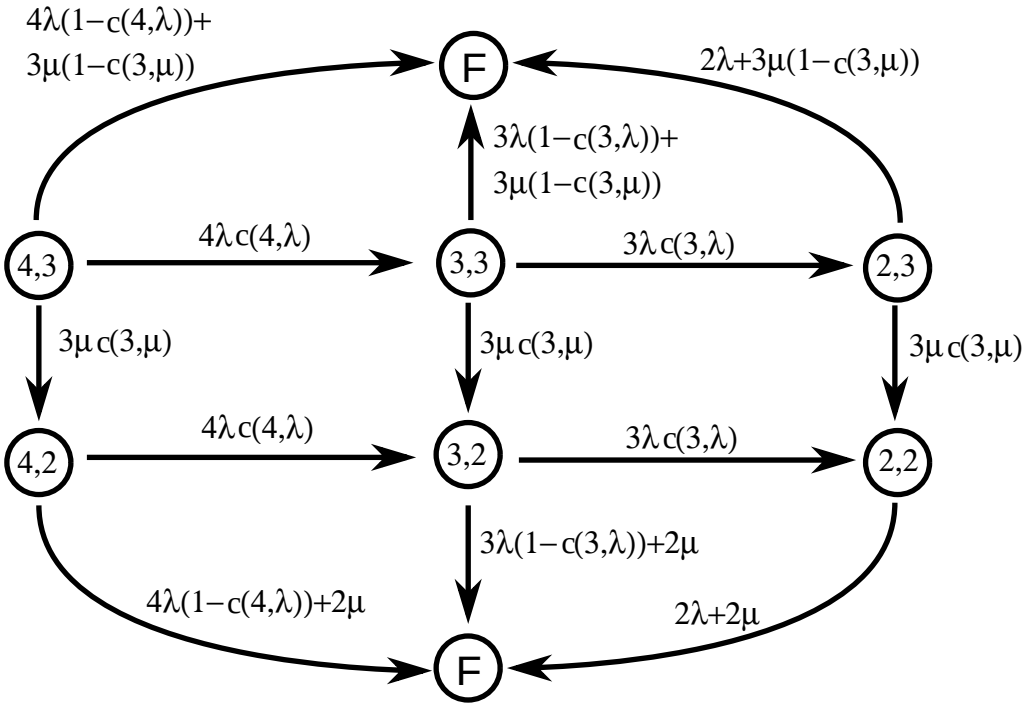


Figure 4.18: A Markov Model for a SIFT-like System

```
1   * SIFT example
2
3   func c(x,k) 360/(360 + (k-1) * x)
4   func r(x,k) k * x * c(x,k)
5   func f(x,k) k * x * (1-c(x,k))
6
7   markov SIFT
8   4-3 3-3 r(lambda,4)
9   3-3 2-3 r(lambda,3)
10
11  4-3 F f(lambda,4) + f(mu,3)
12  3-3 F f(lambda,3) + f(mu,3)
13  2-3 F (2 * lambda) + f(mu,3)
14
15  4-3 4-2 r(mu,3)
16  3-3 3-2 r(mu,3)
17  2-3 2-2 r(mu,3)
18
19  4-2 3-2 r(lambda,4)
20  3-2 2-2 r(lambda,3)
21
22  4-2 F (2 * mu) + f(lambda,4)
23  3-2 F (2 * mu) + f(lambda,3)
24  2-2 F (2 * mu) + (2 * lambda)
25  end
26  end
27
28  bind
29  lambda .0001
30  mu .00001
31  end
32
33  expr mean(SIFT,F)
34  expr variance(SIFT,F)
35  expr value(10;SIFT,F)
36  end
```

Figure 4.19: Input File for the SIFT-like system

```
mean(SIFT,F): 1.0604e+04

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

variance(SIFT,F): 3.9905e+07

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

value(10;SIFT,F): 3.7333e-08
```

Figure 4.20: Results for the SIFT System

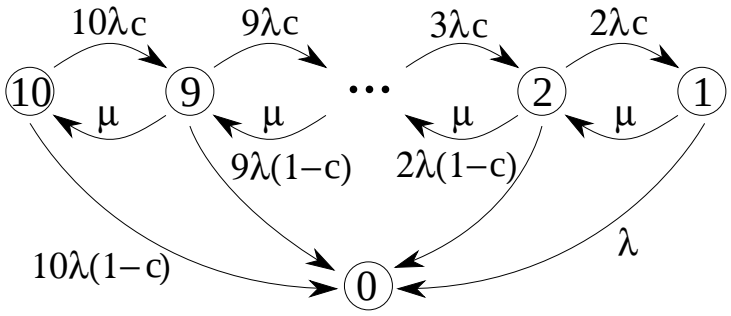


Figure 4.21: A Cyclic Markov Chain with Absorbing States

the time-to-failure of the system. We might be interested in knowing the probability that this system fails within ten hours. On line 35, we use **expr** and the built-in function **value**, which evaluates the CDF for the given state (the third argument) in the given system (second argument) at the given time value (the first argument). Note that the time value is followed by a semicolon. If the system *SIFT* had been defined with parameters, the state name F would have been followed by a semicolon and an argument list. Note that since this system has only one absorbing state (F), the time to reach state F is exactly the same as the time to reach absorption for the system as a whole.

Figure 4.20 shows the results produced by SHARPE for this model.

SHARPE shows that the probability that the system fails within ten hours is .000000037333. Depending on the application of this system, that may or may not be considered small enough.

4.10 A Cyclic Markov Chain with Absorbing State

Suppose we have a system with ten statistically identical components configured in parallel and with a single repair facility. Assume that some faults can not be handled by the automatic detection and reconfiguration mechanism of the system. The system has failed if all components have failed or if a non-recoverable failure occurs in one of the components. A Markov chain for the failure process is shown in Figure 4.21 where state i of the Markov chain models the system state with i operational components. The failure rate for each of the components is given by λ. The repair rate is μ. The coverage factor (probability that a faulty component is detectable and reconfigurable) is c. The 2-component version of this system was used as an example by Arnold [2] and also discussed in [53].

We are interested in how much improvement we get in mean time to system failure because there is a repair facility. If MTTF (μ) is the mean time to system failure when the repair rate is μ, then we define the Mean Time Improvement Factor (MTIF) to be MTIF(μ) = MTTF(μ)/MTTF(0). We would like to know MTIF(μ) for various value of μ and to see how much affect the coverage value c has on the MTIF.

```
markov 10proc(lam, mu, c)
10 9 10*lam*c
10 0 10*lam*(1-c)
9 8 9*lam*c
9 0 9*lam*(1-c)
8 7 8*lam*c
8 0 8*lam*(1-c)
7 6 7*lam*c
7 0 7*lam*(1-c)
6 5 6*lam*c
6 0 6*lam*(1-c)
5 4 5*lam*c
5 0 5*lam*(1-c)
4 3 4*lam*c
4 0 4*lam*(1-c)
3 2 3*lam*c
3 0 3*lam*(1-c)
2 1 2*lam*c
2 0 2*lam*(1-c)
```

```
1 0 lam
1 2 mu
2 3 mu
3 4 mu
4 5 mu
5 6 mu
6 7 mu
7 8 mu
8 9 mu
9 10 mu
end
10 1.0
end
```

```
func mtif(mu) mean(10proc;lam, mu, c) \
    mean(10proc;lam, 0, c)

bind c 0.9999
bind lam 1.0

expr c

expr 1.0/lam, mtif(lam,1.0,c)
expr 5.0/lam, mtif(lam,5.0,c)
expr 10.0/lam, mtif(lam,10.0,c)

bind c 1.0
expr c
expr 1.0/lam, mtif(lam,1.0,c)
expr 5.0/lam, mtif(lam,5.0,c)
expr 10.0/lam, mtif(lam,10.0,c)

end
```

Figure 4.22: Input File for the Ten Processor Example

Figure 4.22 shows an input file for this system. The specification of the Markov chain is as for previous examples. Notice the definition of the function *mtif.* It is defined in terms of the mean of the CDF for the same Markov chain analyzed with two different sets of parameters. It is also worth noting that if c is nonzero, then the Markov chain with the first set of parameters is cyclic and the second is not.

Output is shown in Figure 4.23.

4.11 Irreducible Markov Chains: Comparing Repair Strategies

To show how irreducible Markov chains can be used, consider a system with three statistically identical components, each with failure rate λ. The system is up whenever one or more of the three components are up. When a component fails, it gets repaired. The measure of interest is the system's steady state availability.

Suppose we have a single repair facility with repair rate μ and have found that the availability of the system is not high enough. Suppose we want to evaluate two possible improvements to the system. One is to get two more of the same kind of repair facility and the other is to trade in the single repair facility for one that is twice as fast. single repair facility for a faster one. We are considering three repair strategies:

Scheme 2: A single repair facility of rate μ is shared among all components.

Scheme 1: Each component has its own repair facility with repair rate μ.

Scheme 3: Speedup of the repair facility to rate 2μ, while retaining a single repair facility.

Three Markov chains for this example are shown in Figure 4.25 where state j models a system with j working components. Input and output files for SHARPE for this example is shown in Figure 4.25. Notice that we can use model parameters to let us define a single SHARPE model to serve for all three schemes. The output confirms that schemes 2 and 3 both improve steady-state availability, and that scheme 3 improves it more than scheme 2.

```
c:  9.9990e-01                              c:  1.0000e+00

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

1.0/lam :  1.0000e+00                       1.0/lam :  1.0000e+00
mtif(lam,1.0,c):  1.4133e+00                mtif(lam,1.0,c):  1.4137e+00

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

5.0/lam :  5.0000e+00                       5.0/lam :  5.0000e+00
mtif(lam,5.0,c):  1.3277e+01                mtif(lam,5.0,c):  1.3524e+01

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

10.0/lam :  1.0000e+01                      10.0/lam :  1.0000e+01
mtif(lam,10.0,c):  2.3177e+02               mtif(lam,10.0,c):  4.9550e+02
```

Figure 4.23: Output for the Ten Processor Example

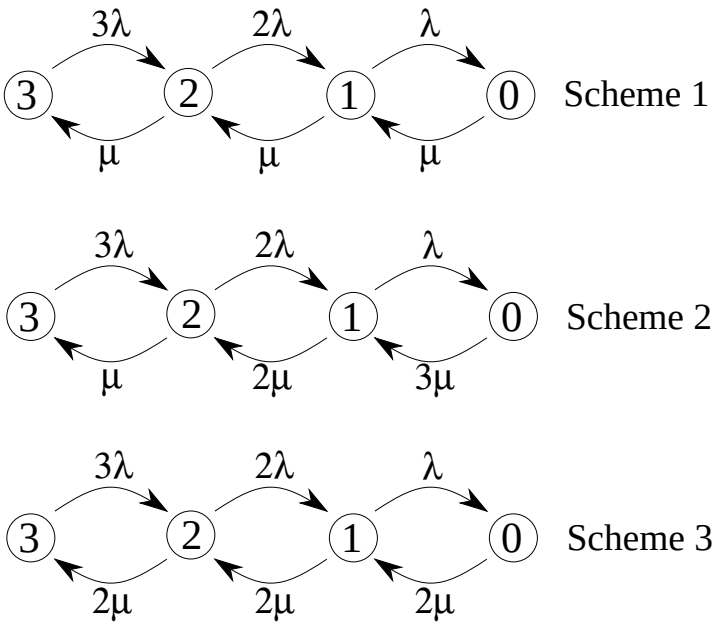


Figure 4.24: Markov Models for the Three Repair Strategies

```
markov 3repair(r1,r2,r3)
3 2 3*lam
2 1 2*lam
1 0 lam
0 1 r1
1 2 r2
2 3 r3
end

bind
lam 0.0001
mu 0.001
end

func avail(r1,r2,r3) \
1 - prob(3repair,0;r1,r2,r3)

var scheme1 avail (mu,mu,mu)
var scheme2 avail (mu,2*mu,3*mu)
var scheme3 avail (2*mu,2*mu,2*mu)

expr scheme1, scheme2, scheme3
end
```

a) input

```
scheme1:  9.9561e-01
scheme2:  9.9910e-01
scheme3:  9.9936e-01
```

b) output

Figure 4.25: Input and Output for Arnold's System

Chapter 5

Performance Analysis

There are several different analytic models that are suitable for perfomance analysis. Each of the models, has its own limitations and advantages over the others:

1. Series-Parallel Directed Acyclic Graphs These can be used to model concurrency and synchronization within programs with unlimited resources. However, contention for limited resources cannot be modeled using such graphs.
2. Product Form Queuing Networks (PFQN): These models are necessary for representation of contention for limited resources. However realistic situations like concurrency within a job, synchronization, simultaneous resource possession etc. cannot be modeled using PFQNs as they violate the assumptions required for an efficient (product-form) solution.
3. Markov chains: These models overcome the inadequacies of the above two model types. They provide a general framework that can be used to model all the above mentioned characteristics of systems. However, the construction of these models can be very difficult and error-prone. Generalized Stochastic Petri Nets provide a higher level interface which can be used for concise description of these models. The underlying Markov chain can then be generated automatically and solved using existing methods.

For our present purpose of performance modeling there are six different model types:

- Series-parallel acyclic directed graphs
- Single or Multiple-chain Product-Form Queuing Networks
- Markov chains
- Semi-Markov chains
- Generalized Stochastic Petri nets
- Any Hierarchical combination involving the above model types.

In this chapter we will provide examples that illustrate the use of above model types for performance analysis.

5.1 Program Performance Analysis

In this section, we consider methods for predicting the performance of programs. For several different methods for analyzing sequential programs, the reader is referred to [53]. For example, if we consider structured programs then we can use combinatorial methods (see chapters 1-5 and appendix E of [53]). For more general programs discrete time Markov chains can be used (see chapter 7 of [53]).

For programs with internal concurrency Bucher has conducted a study of their execution times and speedups on vector and parallel processing systems using measurements [?]. Analytic evaluation of the execution times

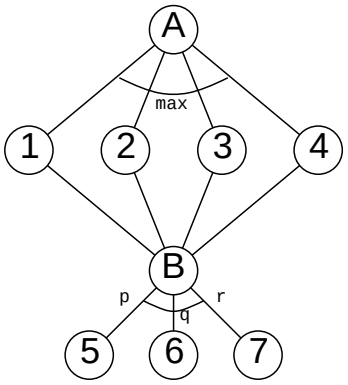


Figure 5.1: A Task Precedence Graph

for programs with internal concurrency can be equated to the analysis of an extended stochastic PERT network [15, 13]. The general problem is quite difficult but it is possible to derive bounds on the performance. Our approach is to consider series-parallel graphs for computing the distribution function of the execution time of a concurrent program in an environment with ample processors. In the case that the actual program graph is not series-parallel and/or it contains cycles, we will see that we can often use the model hierarchies available in our approach to solve the problem.

Series-parallel directed acyclic graphs are very useful in analyzing task graphs of programs. By associating various task graphs with nodes in a SP graph, we are able to evaluate program performance measures such as the distribution of time to completion of the program, dynamic failure probability of the program etc. SP graphs allow the completion time distributions associated with each task to be an arbitrary exponential polynomial. However their only drawback is that they cannot model non-series parallel graphs. The following sections present some examples of program performance using SP graphs.

5.1.1 Performance Analysis of a Concurrent Program

We now consider a model of program execution for a program that has internal concurrency. A task graph for the program is shown in Figure 5.1. To reduce clutter, the distributions for the nodes are not shown. Typically, the program segment A might be responsible for reading data and setting up a number of disjoint subproblems that need to be solved. These problems are passed to a number of processors to be solved concurrently (in this example, we assume four processors). The subproblems might be solved with identical algorithms or with different algorithms. After all of the concurrent segments have finished, segment B collects the answers and examines them. Since node B cannot begin execution until all of the segments 1 through 4 have finished, the finishing time for the collection of segments is the maximum of their individual finishing times. This is expressed in the SHARPE model by assigning an exit type of "maximum" to segment A.

Depending on the results, the program might terminate or one of two possible further actions might be taken. In the graph, nodes 5 and 6 represent the two possible actions. Node 7 is a null action (having distribution zero) and represents the possibility that no action is taken. Since only one of the nodes 5, 6, or 7 is chosen, segment B is assigned the exit type "probabilistic", and the edges going from B to each of 5, 6, and 7 are assigned a probability value.

An input file for this model is shown in Figure 5.2. Lines 3 through 31 define the series-parallel graph. On line 3, the keyword **graph** identifies the system as a series-parallel graph, and the graph is given the name *program*. Lines 4 through 14 define the graph structure by giving one edge on each line. Each edge is specified by its start and end node. The end of the edge definition section is marked by line 15 containing the keyword **end**. Lines 16 through 31 define the concurrency types for parallel subgraphs and the distribution functions for the nodes. In this section, there are three types of lines.

1. exit type assignment (keyword **exit**)

   Nodes that have multiple successors must be assigned exit types. In this example, node A is given exit type **max** on line 16 and node B is given exit type **prob** on line 25.

```
1   * a program with concurrency
2
3   graph program
4   A 1
5   A 2
6   A 3
7   A 4
8   1 B
9   2 B
10  3 B
11  4 B
12  B 5
13  B 6
14  B 7
15  end
16  exit A max
17  dist A exp(uA)
18  dist 1 exp(u1)
19  dist 2 exp(u2)
20  dist 3 exp(u3)
21  dist 4 exp(u4)
22  dist B gen 1,0,0 \
23  -1,0,-2*uA \
24  -2*uA,1,-2*uA
25  exit B prob
26  prob B 5 p
27  prob B 6 q
28  dist 5 exp(u5)
29  dist 6 exp(u6)
30  dist 7 zero
31  end
32
33  bind
34  uA 2
35  u1 1
36  u2 1.4
37  u3 1.2
38  u4 1.1
39  u5 .6
40  u6 .7
41  p .4
42  q .4
43  end
44
45  expr mean(program)
46  expr variance(program)
47  eval (program) 1 10 1
48  end
```

Figure 5.2: Input File for Program with Concurrency

2. distribution assignment (keyword **dist**)

   Every node must be assigned a distribution. In this example, nodes A and 1 through 6 are assigned exponential distributions with different arguments. Node B is assigned a 2-stage Erlang distribution with argument twice that of node A. On line 30, node 7 is assigned the distribution **zero**, which is a built-in distribution type that has no parameters and requires no parentheses.

3. probability values (keyword **prob**)

   Whenever a node has exit type **prob**, the edges leaving the node must be assigned probability values. On lines 26 and 27, we assign p to the edge going from B to 5 and q to the edge going from B to 6. There is no probability assigned to the edge going from B to 7. SHARPE allows us to leave one probability unspecified; it will compute that probability by adding up the other probabilities and subtracting the sum from one. Of course, the partial sum must be less than or equal to one. If all of the probabilities are specified, they must add up to one.

The lines within this section can appear in any order, with the obvious exception that a group of lines continued with "\" cannot be separated from each other.

On lines 33 through 43, we bind the variables to values. Since this is a performance model, the CDF for the graph is that of the execution time of the program. We are often not so much interested in the actual CDF as in its mean and variance and its values at points of interest. On lines 45 and 46, we request the mean and variance of the CDF for the system using the **expr** keyword and the built-in functions **mean** and **variance**. These functions give the mean and variance, respectively, of the CDF for the graph. On line 47, we request that the CDF be evaluated over the interval 1 through 10 at increments of 1. The results are given in Figure 5.3.

5.1.2 Modeling Interprocess Communication

Consider the task graph shown in Figure 5.4(a). This example is based on a model suggested by Kung [30] Nodes 1, 2, 3, and 4 represent tasks; after task 1 is completed, tasks 2 and 3 could be executed concurrently

```
mean(program):  4.0520e+00

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

variance(program):  3.8177e+00

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

     system  program
     t       F(t)

1.0000 e+00 5.9672 e-03
2.0000 e+00 1.0767 e-01
3.0000 e+00 3.3506 e-01
4.0000 e+00 5.7027 e-01
5.0000 e+00 7.4498 e-01
6.0000 e+00 8.5581 e-01
7.0000 e+00 9.2067 e-01
8.0000 e+00 9.5702 e-01
9.0000 e+00 9.7691 e-01
1.0000 e+01 9.8765 e-01
```

Figure 5.3: Results for Program with Concurrency

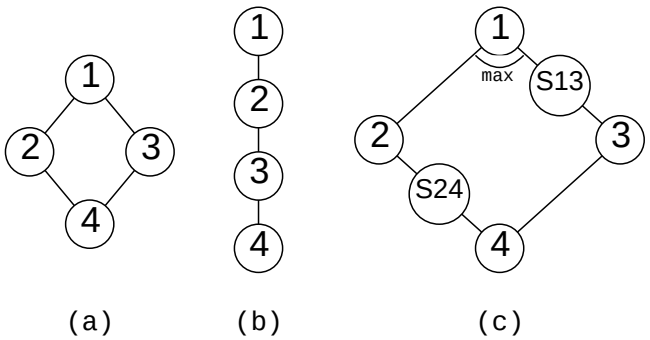


Figure 5.4: Three Task Precedence Graphs

if two processors are available. Since task 4 needs the results of tasks 2 and 3, it cannot begin execution until tasks 2 and 3 are finished. Suppose first that the system on which this task graph is to execute has only one processor and hence it has to be executed sequentially. The graph in Figure 5.4(b) is one way of sequencing the tasks. Next suppose that the system on which the graph of Figure 5.4(a) is to execute possesses two processors. Assume that tasks 1 and 2 are executed on one processor and tasks 3 and 4 are executed on another processor. Results from task 1 must be sent from one processor to the other before task 3 can begin, and similarly for tasks 2 and 4. The time needed for communication between tasks 1 and 3 and tasks 2 and 4 is modeled by the nodes S13 and S24, respectively, in the graph of Figure 5.4(c).

Kung assumed that all of the distributions are exponential and analyzed the graph by converting it into a Markov chain. The SHARPE technique allows distributions to be any exponential polynomial. We assigned task 2 the 2-stage Erlang distribution with parameter 0.4, and the rest of the tasks the exponential distribution with parameters 0.3, 0.57 and 0.25 for tasks 1, 3 and 4, respectively. Each communication task is exponentially distributed with mean c. By varying the value of c, we can get a feel for when the cost of communication outweighs the benefits gained from using two processors. A SHARPE input file for this example is shown in Figure 5.5 (a). In the input file, the graph called *SEQUENCE* models the sequential execution as in Figure

```
poly erlang (u) gen \
   1,0,0 \
   -1, 0, -u \
   -u, 1, -u

bind
u1 .3
u2 .4
u3 .57
u4 .25
end

graph SEQUENCE
1 2
2 3
3 4
end
dist 1 exp(u1)
dist 2 erlang(u2)
dist 3 exp(u3)
dist 4 exp(u4)
end
```

a) input

```
graph PARALLEL(c)
1 2
1 s13
2 s24
s13 3
s24 4
3 4
end
exit 1 max
dist 1 exp (u1)
dist 2 erlang (u2)
dist s13 exp (1/c)
dist s24 exp (1/c)
dist 3 exp (u3)
dist 4 exp (u4)
end

expr mean (SEQUENCE)
expr mean (PARALLEL;.5)
expr mean (PARALLEL;.75)
expr mean (PARALLEL;1)
expr mean (PARALLEL;1.25)
expr mean (PARALLEL;1.5)
end
```

```
mean (SEQUENCE): 1.4088e+01

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

mean (PARALLEL;.5): 1.3155e+01

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

mean (PARALLEL;.75): 1.3433e+01

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

mean (PARALLEL;1): 1.3721e+01

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

mean (PARALLEL;1.25): 1.4017e+01

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

mean (PARALLEL;1.5): 1.4321e+01
```

b) output

Figure 5.5: Input and Output for Kung's Example

```
poly erlang (u) gen \
  1,0,0 \
  -1, 0, -u \
  -u, 1, -u

poly defective (f,u) gen \
  u/(f+u), 0, 0 \
  -u/(f+u), 0, -(f+u)

bind
u1 .3
u2 .4
u3 .57
u4 .25
c 1.0
k13 .0001
k24 .0003
end
```

```
graph PARALLEL(f1,f2)
1 2
1 s13
2 s24
s13 3
s24 4
3 4
end
exit 1 max
dist 1 exp (u1)
dist 2 erlang (u2)
dist s13 defective(f1,1/c)
dist s24 defective(f2,1/c)
dist 3 exp (u3)
dist 4 exp (u4)
end

cdf (PARALLEL;0,0)
cdf (PARALLEL;k13,k24)
end
```

Figure 5.6: Input File for Extended Version of Kung's Example

5.5(b), and the model called *PARALLEL* models the concurrent execution of the task graph as in Figure 5.4(c), including the effect of communication delays. From the results shown in Figure 5.5(b), we see that when c is greater than about 1.25, the communication cost causes the two-processor implementation to take longer (on the average) than if the tasks were all run on a single processor.

We can use the defective distributions allowed by SHARPE to model the case where the communication link can fail so that with some probability, the overall program will not finish. If the link used for communication task S13 has failure rate λ_{13} then the completion time distribution for S13 is $(\frac{1/c}{\lambda_{13}+1/c})(1-e^{-(\lambda_{13}+1/c)t})$. The distribution for S14 is computed similarly. When these defective distributions are used, the resulting CDF for the entire graph is defective, and gives both the probability of a link failure before completion of all the tasks as well as the distribution for the time-to-finish in case all tasks do complete. If $\lambda_{13} = .0001$, $\lambda_{24} = .0003$, and $c = 1$, the probability of a link failure before completion of the graph is 0.0003999. A SHARPE input file for this example is shown in Figure 5.6.

5.1.3 Speedup with Varying Number of Processors

Graph models can be used to compute the speedup achieved by using concurrency within programs. Fig. 5.7(a) shows a parallel program in pseudo-code. Fig. 5.7(b) shows the graph when we have three processors to run these tasks on. The graph converted to show the execution on a single processor is shown in Fig. 5.7(c). In the figure, p denotes the probability that B is true.

If we let X_i denote the random execution time of task T_i, the execution time Y_3 of the program when run on three processors, is given by:

$$Y_3 = \begin{cases} X_1 + X_3 + X_7 + X_8 & \text{if B = false} \\ X_1 + X_2 + \max\{X_4, X_5, X_6\} + X_8 & \text{if B = true} \end{cases}$$

For the sequential case the execution time of the program is given by:

$$Y_1 = \begin{cases} X_1 + X_3 + X_7 + X_8 & \text{if B = false} \\ X_1 + X2 + X4 + X5 + X6 + X8 & \text{if B = true} \end{cases}$$

Two important performance measures can be computed using this model: the speedup and the probability of finishing the job by a particular time. The first is a simple computation of the ratios of the mean execution times

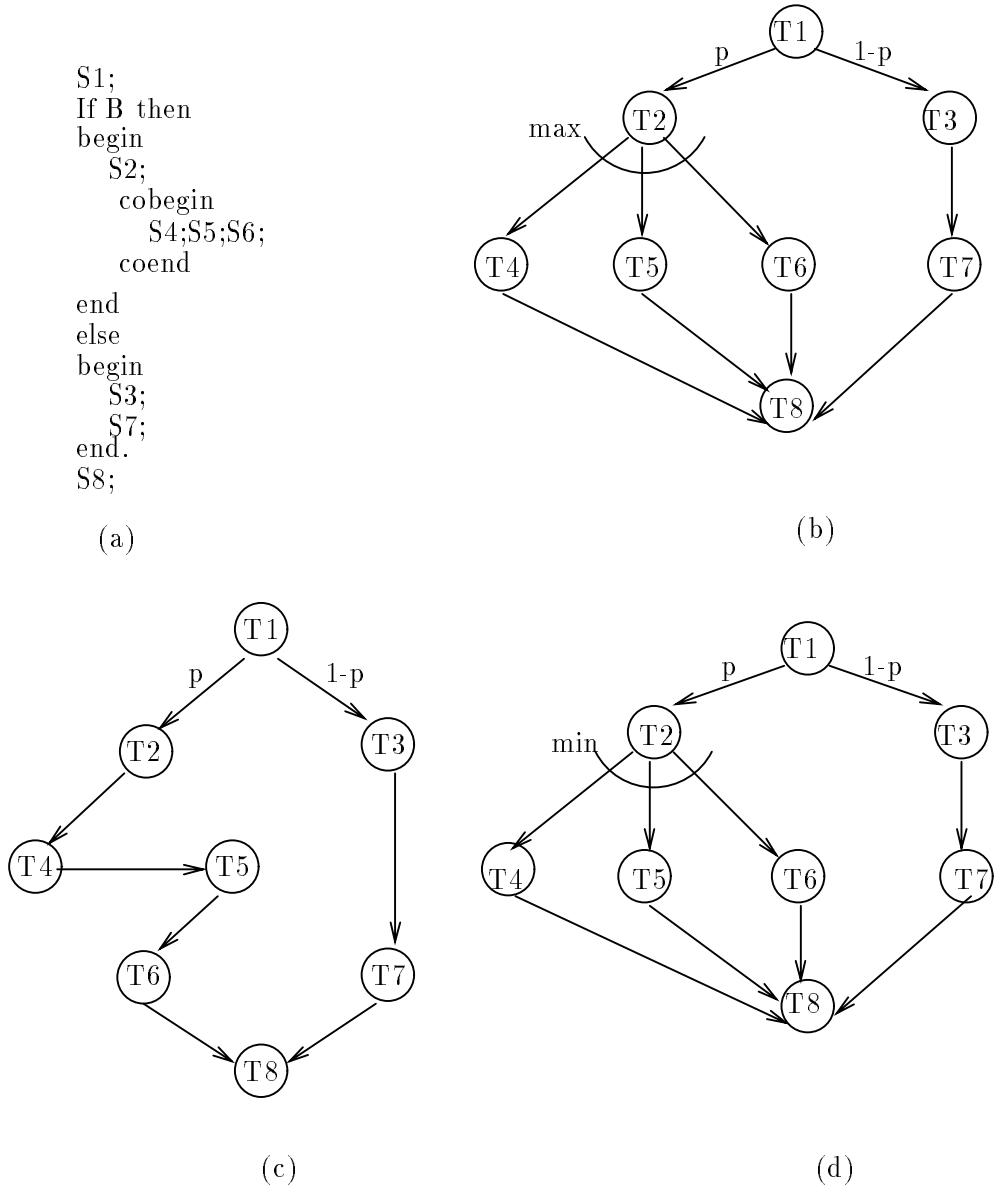


Figure 5.7: (a) Example parallel program. Graph models for the execution of the program on (b) 3 processors, (c) 1 processor, (d)"min" case

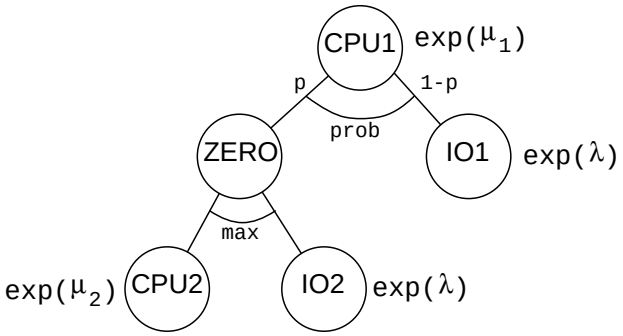


Figure 5.8: Precedence Graph for the CPU-I/O Overlap Example

($E[Y_1]/E[Y_3]$), while the second one is available to us through the CDF (Cumulative Distribution Function) of the completion time of the task graph ($P(Y_3 \leq d)$ or $P(Y_1 \leq d)$). For example, when $T1, T2 \sim EXP(1)$, $T3 \sim EXP(0.2)$, $T7 \sim EXP(0.5)$, $T8 \sim EXP(1.3)$, $T4, T5, T6 \sim EXP(0.1)$ and $p = 0.9$, the speedup achieved is 1.5285. The probability that the program is completed within time $t = 30$ on a single processor is 0.595 which is considerably lower than that for three processors, which is 0.83167.

Now consider the case where the type of concurrency spawned after the completion of task T_2 is such that the first task to complete among T_4, T_5, and $T6$ may allow the execution of T_8 to begin. This will model, for example, the situation where $T4$,$T5$, and $T6$ are carrying out a parallel search of a database. The graph model for this case is as depicted in Fig. 5.7(d). The random completion time in this case is given by

$$Z_3 = \begin{cases} X_1 + X_3 + X_7 + X_8 & \text{if B = false} \\ X_1 + X_2 + \min\{X_4, X_5, X_6\} + X_8 & \text{if B = true} \end{cases}$$

One can compare the completion time distribution for this with the case of *max* with three processors. The probability that the program completes by a deadline $t = 30$ is 0.991, which is the highest among the three cases considered.

5.1.4 CPU/IO Overlap

Figure 5.8 shows a series-parallel graph representing one iteration of the program with CPU-I/O overlap considered by Towsley, Chandy and Browne [52]. 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 input/output, or input/output that may be overlapped with a second CPU burst. The probability that the second stage consists of CPU-I/O overlap is given by p. The node called *ZERO* is a dummy node having distribution **zero**. It allows us to have one branch of the *CPU* node lead to a single node, while the other branch leads to a group of nodes to be executed in parallel.

We might like to know how much it helps to allow the overlap. We define the "speedup" to be the ratio of the mean sequential execution time (the time when no overlap is allowed) to the mean parallel execution time. We can use the SHARPE program to compute the speedup for various values of p. Figure 5.9 shows a SHARPE input file for this example.

Note the use of the **loop** keyword to vary the value of p from 0.6 to 1.0. Note also that we have used an alternate one-line form of the **bind** statement.

Results are shown in Figure 5.10. When p is 0.7, the mean execution time for the serial graph is .27505, the mean execution time for the graph with overlap is .22733, and the speedup is 1.21. It is interesting to note that even when we have maximum parallelism for this graph (when the branch leading to *IO1* is never taken), the speedup is only 1.28. This is because of the time spent in *CPU1*. When we decreased the mean service time at *CPU1* from 0.0376 to 0.01, the speedup with maximum parallelism increased to 1.31.

As a further experiment, we can add more detail to the model. First, we recognize the fact that the length of each CPU burst may vary, depending on the particular job being done. We can model this by dividing the jobs into n classes and assigning each CPU node in the graph an n-stage hyperexponential distribution. Second, we assume that the I/O service consists of three sequential phases. The first phase, corresponding to seek time,

```
* 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 zero
cpu1 io1
zero cpu2
zero io2
end
```

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

expr mean(SERIAL;0.7)
expr mean(OVERLAP;0.7)
loop p, 0.6, 1.0, 0.1
   expr mean(SERIAL;p)/mean(OVERLAP;p)
end

bind mu1 1 / 0.01

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

Figure 5.9: Input for the CPU-I/O Example

```
mean(SERIAL;0.7) : 2.7505e-01

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

mean(OVERLAP;0.7) : 2.2733e-01

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


p=0.600000
    mean(SERIAL;p)/mean(OVERLAP;p) : 1.1845e+00

p=0.700000
    mean(SERIAL;p)/mean(OVERLAP;p) : 1.2099e+00
```

```
p=0.800000
    mean(SERIAL;p)/mean(OVERLAP;p) : 1.2341e+00

p=0.900000
    mean(SERIAL;p)/mean(OVERLAP;p) : 1.2570e+00

p=1.000000
    mean(SERIAL;p)/mean(OVERLAP;p) : 1.2790e+00
------------------------------


mean(SERIAL;1.0)/mean(OVERLAP;1.0) : 1.3145e+00
```

Figure 5.10: Output for the CPU-I/O Example

is assumed to be exponentially distributed with a mass at the origin:

$$F_{seek}(t) = p_{noseek} + (1 - p_{noseek})(1 - e^{-\lambda_{seek}t})$$

Thus with probability p_{noseek} the required information is on the cylinder located under the head, and hence no head movement is necessary. The second phase is the rotational latency phase, and the third phase is the transfer phase; we assume that the time spent in each of these phases is exponentially distributed.

We choose the parameter values so that the mean for the 2-stage hyperexponential distribution for each CPU node is the same as the mean for the previously used exponential distributions for the nodes, and the mean for the I/O nodes is the same as before. The more detailed model shows a shorter mean execution time for the graph with overlap and a greater speedup for each value of p. This illustrates the fact that the mean of a distribution does not contain all of the information about a distribution. A SHARPE input file for the extended case is shown in Figure 5.11.

5.1.5 Program Execution with a Possibility of Failure

To see how SHARPE can be used to analyze the completion time of a program that is subject to software or hardware failure, we consider an example taken from Wei and Campbell [59]. In Figure 5.12, the nodes in the graph represent segments of a process.

Associated with each segment i is the distribution function $F_i(f_i, \mu_i, t) = (1 - f_i)(1 - e^{-\mu_i t})$. The probability of failure during the execution of the segment is $1 - \lim_{t\to\infty} F(f_i, \mu_i, t) = f_i$. All branching in the graph is probabilistic, and the label p_{ij} on the edge leading from node i to node j gives the probability that after the completion of segment i the branch to segment j is taken.

In [59], a formula is given for approximating the overall failure probability. Using the SHARPE technique, the result function F for the overall graph gives the CDF for the completion time of the entire process. The mass at infinity of this CDF gives the probability p that a failure occurs before the whole process completes. The distribution F/p is the CDF of the process completion time given that a failure did not occur.

We used SHARPE to analyze this graph for the same two sets of values for the probabilities on the edges and failure probabilities as in [16]. We assigned the μ_i arbitrary values, since the original example did not contain execution-time parameters.

Figures 5.13(a) and 5.13(b), respectively, contain the input and output files for this example.

5.1.6 A Large Task Graph

As another example of concurrency, we consider the task graph shown in Figure 2.1 adopted from [**?**]. In [**?**] task execution times were assumed to be deterministic; however, here we treat individual task execution times as random variables. The number associated with each node in Figure 2.1 is taken as the mean time to completion of the task associated with that node in seconds. Times to completion of individual tasks are assumed to be exponentially distributed random variables, unless otherwise mentioned. Acyclic series-parallel graph models along with Markov models (to handle cycles in graphs) are used to solve this problem.

In the following subsections, we first analyze the task graph in order to assess the effect of concurrency on pure performance measures under the assumption that the underlying processors on which the task graph executes do not fail. Next we allow for the possibility of a task failing during execution.

Performance Measures for Completely Reliable Systems

Performance models for the execution of this task graph on systems with one, two and three processors are developed. The uniprocessor system performs these tasks $(T_1, \ldots, T_9)$ sequentially. Two different scheduling schemes, schedule 1 and schedule 2, are used for the two-processor case and only one scheduling scheme is studied for the three processor system. Graph models representing these four cases are shown in Figures 2.2a, 2.2b, 2.2c and 2.2d respectively. The word "max" in Figure 2.2 used at branch points shows that the task at which the branches meet again can be executed only upon completion of all required tasks on those branches. Thus the completion time of the task graph in Figure 2.2d is given by

$$T_1 + max(T_3 + T_6, T_2 + max(T_4, T_5) + max(T_7, T_8)) + T_9$$

```
* 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 zero
cpu1 io1
zero cpu2
zero io2
end

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

```
* CPU-I/O overlap
* with hyperexponential
* CPU service
* and 3-stage IO service

poly hyper(x1,x2) gen \
          1, 0, 0 \
        -0.1, 0, -x1 \
        -0.9, 0, -x2

graph IO
seek latency
latency transfer
end

dist seek gen \
          1, 0, 0 \
   -(1-pn), 0, -Lseek

dist latency exp(Llatency)
dist transfer exp(Ltransfer)
end

graph DETAIL(p)
cpu1 zero
cpu1 io1
zero cpu2
zero io2
end

exit cpu1 prob
prob cpu1 zero p
exit zero max
dist cpu1 hyper(mu1a,mu1b)
dist zero zero
dist io1 cdf (IO)
dist cpu2 hyper(mu2a,mu2b)
dist io2 cdf (IO)
end
```

```
bind
mu1a 0.1 / .0156
mu1b 0.9 / .022
mu2a 0.1 / 0.0250
mu2b 0.9 / 0.1
pn .001
Lseek 1 / .05
Llatency 1 / .02
Ltransfer 1 / .08
end

expr mean(IO)
expr mean (DETAIL;0.8)
expr mean (OVERLAP;0.8)
expr mean (DETAIL;0.9)
expr mean (OVERLAP;0.9)

expr mean(SERIAL;0.6) \
   /mean(DETAIL;0.6)
expr mean(SERIAL;0.7) \
   /mean(DETAIL;0.7)
expr mean(SERIAL;0.8) \
   /mean(DETAIL;0.8)
expr mean(SERIAL;0.9) \
   /mean(DETAIL;0.9)
expr mean(SERIAL;1.0) \
   /mean(DETAIL;1.0)
end
```

Figure 5.11: Input File for the Extended CPU-I/O Example

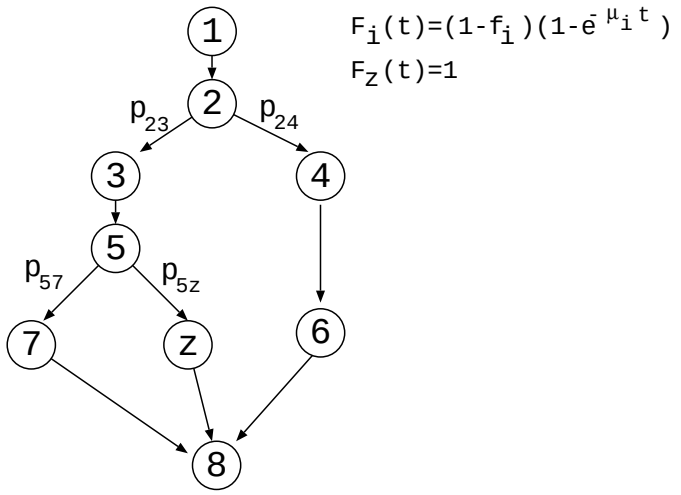


Figure 5.12: A Program Graph

In Figure 2.2d, one processor is scheduled to do tasks T_3 and T_6 while the other two processors are scheduled to tasks on the branch starting at T_2. So the three processors could be simultaneously scheduled to tasks T_6, T_4 and T_5. Note that dummy nodes (such as C and D) with zero completion times are introduced wherever necessary in order to obtain series-parallel graphs without altering the problem.

Two performance measures are used in this analysis, speedup and dynamic failure probability. "Speedup" (S_k) is defined by:

$$S_k = ET_1/ET_k$$

where ET_1 and ET_k are the mean execution times of the task graph on uniprocessor and multiprocessor system with k processors, respectively. Speedup gives us a measure of the benefits of using multiprocessor systems and hence could be used in optimization studies. The second measure "Dynamic Failure Probability at Time d" ($DFP(d)$) is defined as the probability of not completing the job by time, d. Since SHARPE computes the distribution function of the completion time of task graphs, $DFP(d)$ is easily computed. This is a useful measure since it gives us the probability of not meeting a fixed deadline, and thus we are able to assess the timeliness property, an important characteristic of real-time systems.

The results of the analysis obtained from SHARPE are given below in Table 2.1. For the deterministic case with two processors using schedule 1, the speedup given in [?] is 1.8, while for the random case, the speedup is 1.3124 as shown in Table 2.1. As expected, the three-processor case has the best speedup, and for the two-processor case, schedule 2 has better speedup than schedule 1. We use a deadline of 25 seconds for the dynamic failure probability measure. We note that apart from speedup, the use of multiple processors significantly reduces the probability of not meeting the deadline.

Table 2.1 Performance measures for uni- and multi- processor systems

Case	*Speedup*	*DFP(25)*
One processor	1.0	0.6398
Two processors - schedule 1	1.3124	0.37469
Two processors - schedule 2	1.3527	0.34783
Three processors	1.4518	0.29123

Task Failures

Let p_{fi} denote the failure probability of task i. The distribution function assigned to the time to complete task i is then $(1-p_{fi})(1-e^{-\lambda_i t})$, a defective distribution. The combined performance-reliability measure used here is "Dynamic Failure Probability with Task failures"($DFP_{TF}(p,d)$) where p is the probability that a task fails, i.e. $p_{fi} = p \; \forall \, i$. It is defined as the probability that the job is not completed by time d when each task is allowed

```
* program execution with
a possibility of failure

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

graph main
e1 e2
e2 e3
e2 e4
e3 e5
e5 e7
e5 z
e7 e8
z e8
e4 e6
e6 e8
end

exit e2 prob
exit e5 prob
prob e2 e3 p23
prob e5 e7 p57

dist z zero
dist e1 F(f1, u)
dist e2 F(f2, u)
dist e3 F(f3, u)
dist e4 F(f4, u)
dist e5 F(f5, u)
dist e6 F(f6, u)
dist e7 F(f7, u)
dist e8 F(f8, u)
end
```

```
* 1st data set
bind
f1 .02
f2 .03
f3 .04
f4 .05
f5 .06
f6 .07
f7 .08
f8 .09
u 3
p23 .6
p57 .8
end

cdf (main)

* 2nd data set
bind
f1 .002
f2 .003
f3 .004
f4 .005
f5 .006
f6 .007
f7 .008
f8 .009
u 3
p23 .6
p57 .8
end

cdf (main)
end
```

a) input

```
CDF for system main:

probability at 0: 0.0000e+00
probability at infinity: 2.5590e-01
continuous probability: 7.4410e-01

  -6.9806e-01 t( 5) exp(-3.0000e+00 t)
 + -2.5113e+00 t( 4) exp(-3.0000e+00 t)
 + -3.3485e+00 t( 3) exp(-3.0000e+00 t)
 + -3.3485e+00 t( 2) exp(-3.0000e+00 t)
 + -2.2323e+00 t( 1) exp(-3.0000e+00 t)
 +  7.4410e-01 t( 0) exp( 0.0000e+00 t)
 + -7.4410e-01 t( 0) exp(-3.0000e+00 t)

mean and variance are conditional on finite time

mean: 1.8211e+00
variance: 6.3466e-01

_____________________

CDF for system main:

probability at 0: 0.0000e+00
probability at infinity: 2.8319e-02
continuous probability: 9.7168e-01

  -9.4129e-01 t( 5) exp(-3.0000e+00 t)
 + -3.2794e+00 t( 4) exp(-3.0000e+00 t)
 + -4.3726e+00 t( 3) exp(-3.0000e+00 t)
 + -4.3726e+00 t( 2) exp(-3.0000e+00 t)
 + -2.9150e+00 t( 1) exp(-3.0000e+00 t)
 +  9.7168e-01 t( 0) exp( 0.0000e+00 t)
 + -9.7168e-01 t( 0) exp(-3.0000e+00 t)

mean and variance are conditional on finite time

mean: 1.8261e+00
variance: 6.3643e-01
```

b) output

Figure 5.13: Input and Output Files for Program Execution with Failure

to fail with probability p. Table 2.2 shows the value of this measure for one- and three- processor systems with $p = 0.1$ and $d = 25$ *seconds*. This table has been included so that the reader could compare these values with those of Table 2.1 to see the effect of allowing task failures.

Table 2.2 Combined measure for one- and three- processor systems

Case	$DFP_{TF}(0.1, 25)$
One processor	0.86045
Three processors	0.72541

Fig. 2.6 shows plots of the variation of $DFP_{TF}(p, d)$ with respect to d for both one and three processor systems and for three values of p, 0, 0.1 and 0.5. When $p = 0$ DFP_{TF} reduces to DFP as defined in the previous section. Note that for $p = 0.5$, the values of DFP_{TF} remains the same for one and three processors for the time range considered.

Another useful measure that can be obtained from this analysis is the probability that the job never completes also called the omission failure probability ($OFP(p)$). This can happen since tasks are allowed to fail and not recover. The omission failure probability is obtained directly in SHARPE by requesting the mass at infinity in the overall execution time distribution for the task graph. This obviously depends on the value of p as shown below in Table 2.3.

Table 2.3 Omission failure probability

p	*OFP - 1 processor*	*OFP - 3 processors*
0.0	0	0
0.1	6.1258e-01	6.1258e-01
0.5	9.9805e-01	9.9805e-01

Effect of Different Distributions for Task Completion Times

We then test the effect of assigning distributions other than exponential for the completion times of tasks. The task graph used is that in Figure 2.1 and no task or server failures are considered. The servers used are the one and three processor systems with scheduling as in Figures 2.2a and 2.2d. Performance measures are speedup and $DFP(25\ seconds)$ as defined in section 2.1. Results are shown in Table 2.6. For each case in Table 2.6, only one node's distribution is explicitly mentioned. All other nodes in the graph models are assumed to have exponentially distributed times to completion unless otherwise mentioned in the Table 2.6. The mean task completion times have been maintained the same while changing distributions and the values used for the mean times are as shown in Figure 2.1. The distributions used are exponential, hypoexponential with two stages and hyperexponential with two phases with branch probabilities of 0.25 and 0.75.

Table 2.6 Effect of varying distributions of some task times

Case	*Speedup*	*DFP(25) - 1 processor*	*DFP(25) - 3 processor*
Exp. distr. for T1	1.4518	0.63980	0.29123
Hypoexp. distr. for T1	1.4518	0.63986	0.29080
Hyperexp. distr. for T1	1.4518	0.63902	0.29295
Exp. distr. for T3	1.4518	0.63980	0.29123
Hypoexp. distr. for T3	1.4519	0.63986	0.29121
Hyperexp. distr. for T3	1.4507	0.63902	0.29145
Exp. distr. for T7	1.4518	0.63980	0.29123
Hypoexp. distr. for T7	1.4753	0.65706	0.27037
Hyperexp. distr. for T7	1.4152	0.59617	0.28928
Exp. distr. for T9	1.4518	0.63980	0.29123
Hypoexp. distr. for T9	1.4518	0.63986	0.29080
Hyperexp. distr. for T9	1.4518	0.63902	0.29295

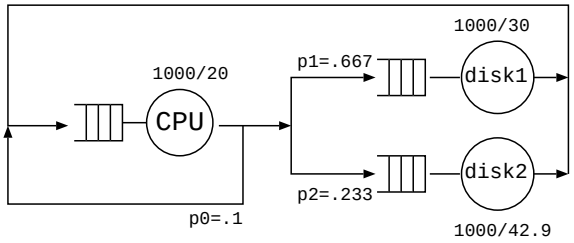


Figure 5.14: The Central-Server Queueing System

The results in Table 2.6 show that if the node that is assigned these various distributions (hypoexponential and hyperexponential) is one that cannot be performed concurrently with any other task (T_1 and T_9 in Figure 2.1), then speedup is insensitive to the distributional assumption. Otherwise, the hypoexponential case gives the most speedup since this distribution has a smaller coefficient of variation than either the exponential or the hyperexponential. Changing the distributions of completion times of nodes T_1, T_3 and T_9 have the same effect on $DFP(25)$ for the one processor system while we get different values for this parameter by changing the distribution of the completion time of node T_7. This is because tasks T_1, T_3 and T_9 have the same mean time to completion of unity while T_7 has a mean time to completion of 7. Hence its contribution to the distribution of the total completion time is different.

5.2 System Performance Analysis

5.2.1 A Central-Server Queueing System

Figure 5.14 shows a "closed central-server model" ([53]), in which jobs receive service at a CPU, then proceed to wait for service from one of two disks or another time-slice from the CPU. The system is assumed to contain a fixed number, N, of jobs. Each server has exponentially distributed service time, with the service rate shown above or below the server.

Figure 5.15(a) shows the SHARPE input for this model. The queueing network model is specified in three sections starting on line 8. The first section (lines 9 through 13) gives the shape of the network and the routing probabilities. Each line is a 3-tuple consisting of two station names and the probability that a job goes to the second station after it has received service at the first.

The second section (lines 15 through 18) specifies the service parameters for the stations. There is a line for each station; the line contains the station name, the service type, and one or more service parameters, the number of parameters depending on the service type. In this case, all of the servers are "first-come-first-serve" **fcfs**; this service type takes one parameter, the service rate.

The third section (lines 20 through 21) gives the number of jobs in the network. It consists of a line containing any identifier followed by the number of jobs. The first identifier on the line is meaningless for a "single-chain" network like this one; for a multiple-chain network there would be a line for each chain, with the first identifier on each line identifying the chain.

SHARPE provides four measures (all steady-state) for each server in a PFQN: throughput, utilization, response time and queue length. In this case, we vary the number of jobs from 2 to 10 to see how the number of jobs affects these measures at the CPU. The results are shown in Figure 5.15(b). We see that average queue length at the CPU grows from 0.8233 to 4.595, average response time at the CPU grows from 0.03 to 0.102, average utilization of the CPU grows from 0.588 to 0.899, and CPU throughput grows from 2.94 to 4.499.

Besides **fcfs**, SHARPE allows the following service types: processor-sharing, multiple, infinite (simultaneously serves all jobs at the same rate), load-dependent, and last-come, first-serve preemptive resume. See Section A.4.6 for more details.

5.2.2 A Terminal-Oriented System

Consider a terminal-oriented system as shown in Fig. 5.16, where each one of the M terminals can issue requests at a rate λ. A request first requires CPU service for an exponentially distributed amount of time. The

```
 1  * central-server queueing system
 2
 3  bind
 4  p1 0.667
 5  p2 0.233
 6  end
 7
 8  pfqn csm
 9  cpu drum p1
10  cpu disk p2
11  drum cpu 1
12  disk cpu 1
13  end
14  * fcfs servers
15  cpu fcfs 1000/20
16  drum fcfs 1000/30
17  disk fcfs 1000/42.918
18  end
19  * number of jobs
20  chain1 custs
21  end
22
23  loop i,2,10,2
24      bind custs i
25      expr tput(csm,cpu)
26      expr util(csm,cpu)
27      expr qlength (csm,cpu)
28      expr rtime (csm,cpu)
29  end
30
31  end
```

a) input

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

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

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

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

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

b) output

Figure 5.15: Input and Output for Central Server Queueing Network

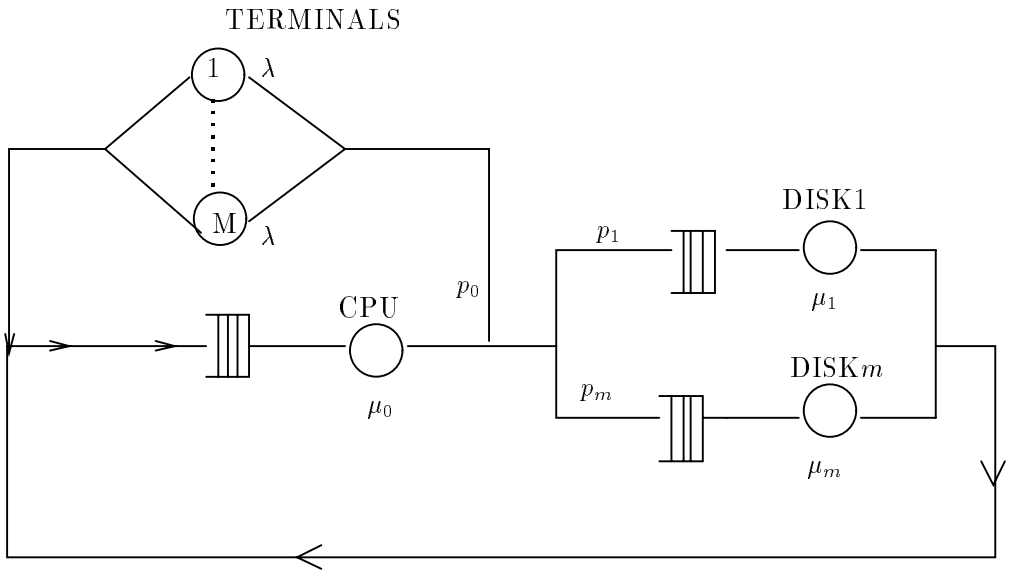


Figure 5.16: Terminal Oriented System

request may then complete with probability p_0 or use either one of the m disks with probability p_i for DISKi. After the use of the disks, the cycle continues from the CPU. If we assume that all the servers have exponentially distributed service times and follow the FCFS scheduling discipline, we have a product-form queuing network. SHARPE's PFQN model type can be used to compute the average performance measures such as throughput and mean response time of the system. If we have $m = 2,\ p_1 = 0.667,\ p_2 = 0.233,\ \lambda = 1/25,\ \mu_0 = 1000/20,\ \mu_1 = 1000/30,\ \mu_2 = 1000/42.918$, the throughput of the CPU is obtained as 1.9597, the mean response time at DISK2, 0.043599, and the mean queue length at DISK1, 0.040473.

5.2.3 Multi-Chain Product Form Queueing Networks

Queueing networks considered thus far were "single-chain" queueing networks. By "single-chain", we mean that all jobs in the queue behave the same with regard to routing probabilities and service characteristics at the stations. If we want the chain to be populated by jobs with different behavior, we can use a "multi-chain" network.

First consider a variation of this example with two classes of jobs : the *CPU-intensive* class and the *I/O-intensive*. Parameters like routing probabilities, service time distributions at the CPU (where PS discipline is used in this case) can differ for these two classes. SHARPE's multiple class PFQN model can be used to compute steady-state performance measures for this case as well. Let $p_0 = 0.7,\ p_1 = 0.1,\ p_2 = 0.2,$ and $\mu_0 = 30$ for the CPU-intensive class, and let $p_0 = 0.2,\ p_1 = 0.5,\ p_2 = 0.3,$ and $\mu_0 = 70$ for the I/O-intensive class. Let the rest of the parameters be the same as before for both the classes. Then, for 2 CPU-intensive jobs and 3 I/O-intensive jobs, the average throughput at the CPU for the CPU-intensive class is 0.11399 and for the I/O-intensive class, 0.0084992.

Next suppose we have a system with two processors, each having one disk, and one shared disk. Suppose that processes in this system are targeted to a particular processor. We can draw this queueing network as shown in Figure 5.17.

Jobs in chain 1 visit processor **P1**, then go either to **P1**'s private disk (**D1**), the shared disk (**Ds**), or back to **P1** (if it left **P1** because the job was finished or reached the end of a time slice). Jobs in chain 2 behave similarly, but use **P2**, **D2** and **Ds**.

A SHARPE input file for this model is shown in Figure 5.18(a). On lines 1 through 10, we bind values to the routing probabilities and the station service rates. For the sake of illustration, we decided to assume that the

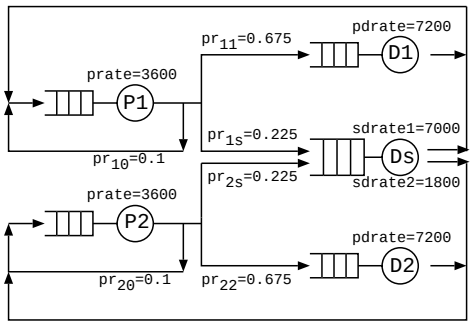


Figure 5.17: A Multi-Chain Queueing Network

```
1    bind
2    pr11 .675
3    pr1s .225
4    pr22 .675
5    pr2s .225
6    prate 3600
7    pdrate 7200
8    sdrate1 7000
9    sdrate2 1800
10   end
11
12   mpfqn serve2(c)
13   chain 1
14   P1 D1 pr11
15   P1 Ds pr1s
16   D1 P1 1
17   Ds P1 1
18   end
19
20   chain 2
21   P2 D2 pr22
22   P2 Ds pr2s
23   D2 P2 1
24   Ds P2 1
25   end
26   end
```

```
27
28   P1 fcfs prate
29   end
30   P2 fcfs prate
31   end
32   D1 fcfs pdrate
33   end
34   D2 fcfs pdrate
35   end
36   Ds ps
37   1 sdrate1
38   2 sdrate2
39   end
40
41   end
42   1 c/2
43   2 c/2
44   end
45
46   loop c,10,40,10
47     expr mqlength (serve2,Ds; c)
48     expr mqlength (serve2,Ds,1; c),
49          mqlength (serve2,Ds,2; c)
50   end
51
52   end
```

a) input

```
c=10.000000
   mqlength (serve2,Ds; c): 2.8807e+00
   mqlength (serve2,Ds,1; c): 7.7162e-01
   mqlength (serve2,Ds,2; c): 2.1091e+00

c=20.000000
   mqlength (serve2,Ds; c): 7.0994e+00
   mqlength (serve2,Ds,1; c): 1.8135e+00
   mqlength (serve2,Ds,2; c): 5.2859e+00

c=30.000000
   mqlength (serve2,Ds; c): 1.2108e+01
   mqlength (serve2,Ds,1; c): 3.0067e+00
   mqlength (serve2,Ds,2; c): 9.1008e+00

c=40.000000
    mqlength (serve2,Ds; c): 1.7661e+01
    mqlength (serve2,Ds,1; c): 4.3077e+00
    mqlength (serve2,Ds,2; c): 1.3353e+01
```

b) output

Figure 5.18: Input and Output for Multi-Chain Queueing Network

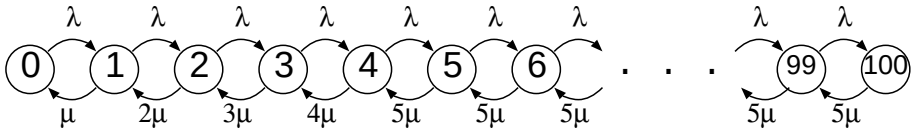


Figure 5.19: A Birth-Death Type Markov Chain for the M/M/5/100 System

shared disk provides asymmetric service; it provides faster service (at **sdrate1**) for jobs running on processor 1 than for jobs running on processor 2 (**sdrate2**).

On lines 12 through 44 we define the multi-chain queueing network. We start with the keyword **mpfqn** on line 12. On lines 13 through 26, we define the shape and routing probabilities of the network, with chain 1 defined on lines 13 through 18 and chain 2 defined on lines 20 through 25. On lines 28 through 41, we define the service characteristics of the stations. Each station has its own section ending with the keyword **end**. This is because except for the station type **fcfs**, it is possible to assign different service parameters for each chain. The service characteristics for **P1** (service type **fcfs** and parameter **prate**) are on lines 28 through 29. Those for **P2** are on lines 30 and 31, for **D1** on lines 32 and 33, and for **D2** on lines 34 and 35. Again for the sake of illustration, we decided to give the shared disk a service type of **ps** (processor-sharing) so we could show how different service parameters could be assigned for each chain. On line **36**, we define just the service type **ps** for **Ds**. This is followed by a line for each chain (lines 37 and 38), with each line containing a chain name followed by the service parameters for the chain, in this case just a single service rate per chain.

It is also possible to specify default service parameters on the line containing the service type. In that case, the line defining the service type could be followed by lines for only those chains for which the service parameters are different than the default parameters. In this case, we could have replace lines 36 through 39 with the lines

```
Ds ps sdrate1
2 sdrate2
end
```

On lines 42 through 44, we specify the number of jobs in each chain. On lines 46 through 50, we use a loop to look at the average queue length for the shared disk as the number of jobs in the network increases from 10 to 40. If the built-in function **mqlength** has two arguments before the semi-colon, the result is the average number of jobs from all chains in the queue. If the function has three arguments before the semi-colon, the third argument is a chain name and the result is the average number of jobs in the queue from that chain.

Figure 5.18(b) shows the results.

5.2.4 The M/M/N/K Queueing System

We consider a queueing system with N processors. We will assume that the arrival stream of jobs forms a Poisson process with rate λ and that the service times of jobs are independent and identically distributed exponential random variables with service rate μ. Further assume that the maximum queue length is K. Such a queueing system is known as an M/M/N/K queue. A birth-death Markov chain for $N = 5$ and $K = 100$ is shown in Figure 5.19. SHARPE input and output files for this Markov model are shown in 5.20.

Using the parameters $\lambda = 1.0$ and $\mu = 1.2$, we request the values of two expressions: the probability that the service facility is idle (**prob**(*mm5,0*)) and the rate at which jobs are turned away (rejected) because the system is full (λ * **prob**(*mm5,100*)). The output produced by SHARPE is 0.43457 and 0, respectively. This tells us that at this level of traffic, the processor is busy less than half the time and it is very unlikely for the backlog to build up beyond 100 jobs.

This input file shows the need for providing some kind of looping mechanism within the SHARPE input language. Such a mechanism would make it easy to specify large systems having a regular structure. Plans for a general loop mechanism are under way.

5.2.5 The M/M/N/K Queueing System with Failure and Repair

```
bind
LAM 1.0
MU 1.2
end

markov mm5
0 1 LAM
1 0 MU
1 2 LAM
2 1 2*MU
2 3 LAM
3 2 3*MU
3 4 LAM
4 3 4*MU
4 5 LAM
5 4 5*MU
5 6 LAM
6 5 5*MU
6 7 LAM
7 6 5*MU
7 8 LAM
8 7 5*MU
8 9 LAM
9 8 5*MU
9 10 LAM
10 9 5*MU
10 11 LAM
11 10 5*MU
11 12 LAM
12 11 5*MU
12 13 LAM
13 12 5*MU
13 14 LAM
14 13 5*MU
14 15 LAM
15 14 5*MU
15 16 LAM
16 15 5*MU
16 17 LAM
17 16 5*MU
17 18 LAM
18 17 5*MU
18 19 LAM
19 18 5*MU
19 20 LAM
20 19 5*MU
20 21 LAM
21 20 5*MU
21 22 LAM
22 21 5*MU
22 23 LAM
23 22 5*MU
23 24 LAM
24 23 5*MU
24 25 LAM
25 24 5*MU
25 26 LAM
26 25 5*MU
26 27 LAM
27 26 5*MU
27 28 LAM
28 27 5*MU
28 29 LAM
29 28 5*MU
29 30 LAM
30 29 5*MU
30 31 LAM
31 30 5*MU
31 32 LAM
32 31 5*MU
..
..
..
..
..
..
..
..
85 86 LAM
86 85 5*MU
86 87 LAM
87 86 5*MU
87 88 LAM
88 87 5*MU
88 89 LAM
89 88 5*MU
89 90 LAM
90 89 5*MU
90 91 LAM
91 90 5*MU
91 92 LAM
92 91 5*MU
92 93 LAM
93 92 5*MU
93 94 LAM
94 93 5*MU
94 95 LAM
95 94 5*MU
95 96 LAM
96 95 5*MU
96 97 LAM
97 96 5*MU
97 98 LAM
98 97 5*MU
98 99 LAM
99 98 5*MU
99 100 LAM
100 99 5*MU
end

var Pidle prob(mm5,0)
var Pfull prob(mm5,100)
var Lreject LAM * Pfull
expr Pidle
expr Lreject
end
```

a) input

```
Pidle: 4.3457e-01
————————————
Lreject: 0.0000e+00
```

b) output

Figure 5.20: Input and Output Files for the M/M/5/100 Queue

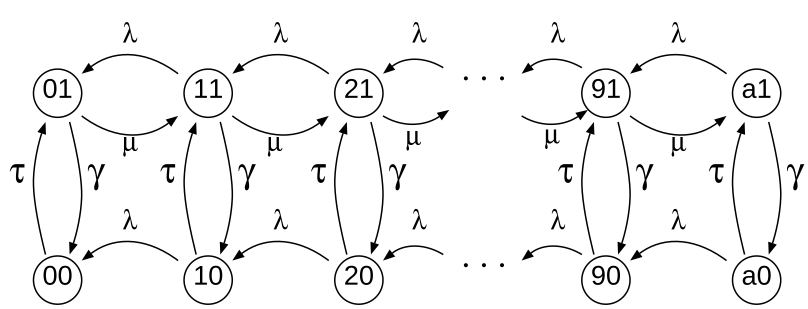


Figure 5.21: Markov Model for the Queueing System with Failure and Repair

```
markov mm1k
01 11 LAM
11 01 MU
11 21 LAM
21 11 MU
21 31 LAM
31 21 MU
31 41 LAM
41 31 MU
41 51 LAM
51 41 MU
51 61 LAM
61 51 MU
61 71 LAM
71 61 MU
71 81 LAM
81 71 MU
81 91 LAM
91 81 MU
91 a1 LAM
a1 91 MU
01 00 GAM
00 01 TAU
11 10 GAM
10 11 TAU
21 20 GAM
20 21 TAU
31 30 GAM
30 31 TAU
41 40 GAM
40 41 TAU
51 50 GAM
50 51 TAU
61 60 GAM
60 61 TAU
71 70 GAM
70 71 TAU
81 80 GAM
80 81 TAU
91 90 GAM
90 91 TAU
a1 a0 GAM
a0 a1 TAU
00 10 LAM
10 20 LAM
20 30 LAM
30 40 LAM
40 50 LAM
50 60 LAM
60 70 LAM
70 80 LAM
80 90 LAM
90 a0 LAM
end

bind
LAM 1
MU 2
GAM 0.0001
TAU 0.1
end
var Pidle prob(mm1k,00) \
      +prob(mm1k,01)
var Pfull prob(mm1k,a0) \
   +prob(mm1k,a1))
var Lreject LAM*Pfull
expr Pidle
expr Lreject
end
```

a) input

```
Pidle: 4.9953e-01
---------------
Lreject: 9.6239e-04
```

b) output

Figure 5.22: Input and Output for the Queueing System with Failure and Repair

Now we consider an M/M/N/K queueing system where the processors are subject to failure, and are repaired whenever they fail. Again, let the job arrival rate be λ and the job service rate be μ. Let the the processor failure rate be γ and the processor repair rate be τ.

First, we model this system by an irreducible Markov chain for $N = 1$ and $K = 10$ (see Figure 5.21). Each state is named with a two-digit number ij where $i \in \{0, 1, ..., 9, a\}$ is the number of jobs in the system and $j \in \{0, 1\}$ is the number of operational processors.

SHARPE input and output files for this example are given in Figure 5.22. The probability that there are no jobs in the system is given by **prob**(*mm1k,00*) + **prob**(*mm1k,01*), and the rate at which jobs are turned away because the system is full is given by $\lambda \times$ (**prob**(*mm1k,a0*) + **prob**(*mm1k,a1*)).

If we wanted to use Markov models to get results for different values of N and K, we would have to build a new Markov model for each different pair of values. Instead, we can use the Generalized Stochastic Petri Net (GSPN) in Figure 5.23 to model the queueing system.

The cycle in the upper part of the figure is a representation of an M/M/1/K queue. The lower cycle models a server that can fail and be repaired. The inhibitor arc from place **server-down** to transition **service** reflects the fact that customers cannot be served while the server is not functioning. The number below each place is

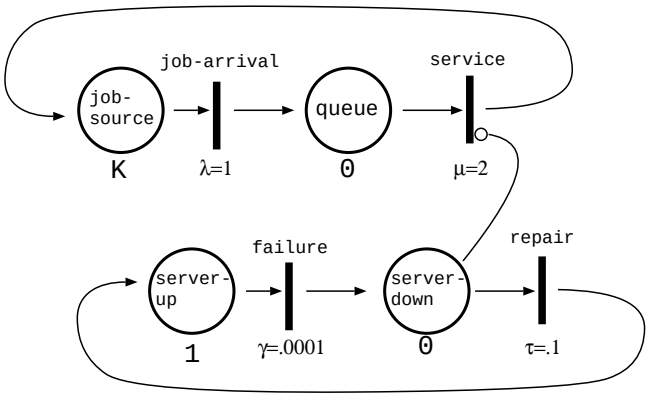


Figure 5.23: Petri Net Model for the Queueing System with Failure and Repair

the initial number of tokens in the place. All of the transitions are timed, and each transition's rate is shown below the transition. An input file for this model is shown in figure 5.24.

On lines 1 through 7, we assign values to values to the model parameters. Starting with line 12, we specify the GSPN in six sections. The first section (lines 15 through 19) specifies the places in the net. Each line is an ordered pair consisting of a place name followed by the number of initial tokens in the place. The second section (lines 21 through 25) specifies the timed transitions. Each line gives the name of a transition followed by the transition type and the transition rate. The transition type is either "independent" (**ind**) or "dependent" **dep**) on the number of tokens on incoming arcs. If the rate is independent, the transition rate gives, unconditionally, the firing rate of the transition. If the rate is dependent, the firing rate is the transition rate multiplied by the number of tokens on incoming arcs. The third section (in this model just the **end** on line 28) gives immediate transitions.

The fourth section (lines 31 through 35) describes the enabling arcs going from places to transitions. Each line is a 3-tuple consisting of a place name, a transition name, and the number of tokens needed to fire the transition. The fifth section (lines 38 through 42) describes the arcs going from transitions to places. Each line is a 3-tuple consisting of a transition name, a place name, and the number of tokens that move to the place when the transition is fired.

The sixth section (lines 44 through 45) gives the inhibitor arcs for the model. Each line is a 3-tuple consisting of a place name, a transition name, and the number of tokens that inhibit the transition from firing when they are in the place.

This GSPN is irreducible. SHARPE makes available four steady-state measures for irreducible GSPNs: throughput, utilization, average number of tokens, and probability of being empty. The first two apply to transitions, the last two to places. On line 50 of figure 5.24, we define the variable **Pidle** to be **prempty (mm1k-fail,queue)**, the probability that the place called **queue** is empty. This is the probability that the server is idle. On line 54, we define **Preject**, the probability that an incoming job is rejected. This happens if the place called **jobsource** is empty (because all **K** jobs are in the place **queue**. On line 57, we define the rejection rate **Lreject**, which is the arrival rate **lambda** multiplied by the probability that a job is rejected. On line 61 we define **avqueulenth**, the average queue length at the server; this is the average number of tokens in the place **queue**. On line 64 we define **thruput** to be the throughput of the transition **service** and on line 67 we define **utilization** to be the utilization of the transition **service**. On lines 69 through 72 we ask for the values of the variables we have defined. Figure 5.25 shows the results.

We can check that the values of **Pidle** and **Lreject** are the same as computed using the Markov chain model (see figure 5.22).

For GSPNs that are non-irreducible, SHARPE also can compute the expected number of tokens in a place at a particular time t, the probability that a place is empty at t, the throughput and utilization of a transition at t, the time-averaged number of tokens in a place during the interval $(0,t)$, and the time-averaged throughput of a transition during $(0,t)$. See table A.2 for the syntax for these functions.

```
1   bind
2   lambda 1.0
3   mu 2.0
4   gamma 0.0001
5   tau 0.1
6   K 10
7   end
```

```
46  * use variables to define some measures
47
48  * of interest
49  * probability that the server is idle
50  var Pidle prempty(mm1k-fail,queue)
51
52  * probability that a job is rejected
```