Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

PERFORMABILITY CONCEPTS AND MODELING TECHNIQUES FOR REAL-TIME SOFTWARE

A. Tai

December 1991
CSD-910080

UNIVERSITY OF CALIFORNIA

Los Angeles

Performability Concepts and Modeling Techniques for Real-Time Software

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Ann Tsu-Ann Tai

1991

The dissertation of Ann Tsu-Ann Tai is approved.

[illegible]

John F. Meyer

Kirby A. Baker

Kirby A. Baker

Thomas Ferguson

Thomas Ferguson

Jack W. Carlyle

Jack W. Carlyle

[illegible]

David A. Rennels

[illegible]

Algirdas Avižienis, Committee Chair

University of California, Los Angeles

1991

In loving memory of my grandmother,

who raised me up

and was always there when I needed her.

She gave me her unconditional support,

and I miss her dearly, forever . . .

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Basic Definitions and Associated Terminology 2

1.3 Organization of the Dissertation 4

2 Prior Work and its Relation to our Research **5**

2.1 A Brief Retrospective 5

2.2 Application of Performability Concepts to Software 6

3 Framework of Software Performability Modeling **10**

3.1 Total System 11

3.2 Performance Variable 14

4 SAN Realization of the Framework **15**

4.1 Representation of the Total System 16

4.2 Example Analysis 20

4.2.1 Software Utilization, Success Criteria, and Expected Reward 20

4.2.2 Coverage and Computational Dependability 24

4.2.3 Workload and Software Failures 31

4.2.4 Maintenance Scheduling 34

5 Refinement of the Framework: a Reward Structure **41**

5.1 Difficulties in Performance Variable Formulation 41
5.2 Evaluating Unified Measures via Distinction 44
5.3 A Reward Structure Using Composite Approach 46
5.4 Discussion 49

6 Applications to Fault-Tolerant Software 51

6.1 Background 51
6.2 Assumptions 52
6.3 Definition of Performance Variable 53
6.4 A Hierarchical Approach to Model Construction and Solution . . 55
6.5 RB Model 57
6.5.1 Dependability Sub-Model 59
6.5.2 Performance Sub-Model 63
6.5.3 Performability Model 65
6.6 NVP Model 69
6.6.1 Dependability Sub-Model 70
6.6.2 Performance Sub-Model 75
6.6.3 Performability Model 76
6.7 Results Analysis 77
6.7.1 Comparisons between RB and NVP 77
6.7.2 Comparisons between Fault-Tolerant Software and Non-Fault-Tolerant Software 82
6.8 Alternative NVP Model 84

6.8.1 An Alternative Approach to NVP Operation 84

6.8.2 Dependability Sub-Model 86

6.8.3 Performance Sub-Model 92

6.8.4 Performability Model 95

6.9 Discussion 96

6.10 Evaluation for Fault Tolerant Software of Closed Loop Type . . . 102

7 Conclusions and Future Work 112

7.1 Reward Structure 112

7.2 Hierarchical Approach 113

7.3 How the Reward Structure and Hierarchical Approach relate to SANs 114

7.4 Summary of Future Work 116

A Glossary of Notation 117

B SAN Structure 121

C Sample C Code for Software Performability Analysis 124

C.1 Program for Computing Expected Reward 124

C.2 Program for Solving a Markov Model for Steady State 127

D Sample Mathematica Program and Output 131

D.1 Mathematica Code Computing the Expected Reward of NVP . . 131

D.2 Mathematica Output for the Expected Reward of NVP 133

D.3 Mathematica Code Computing the Expected Reward of New-NVP 138

D.4 Mathematica Output for the Expected Reward of New-NVP . . . 141

References **147**

List of Figures

3.1 An Overview of the Total System 12

4.1 Example of a Total System 16

4.2 A SAN Model for Total System 17

4.3 Gate Specification for Total System 18

4.4 A SAN Model for a Sampled Data System 21

4.5 Gate Specification for the Sampled Data System 21

4.6 Underlying Markov Chain for the Sampled Data System 21

4.7 Expected Reward as a Function of Invocation Rate (I) 23

4.8 Expected Reward as a Function of Invocation Rate (II) 24

4.9 An Example System Using "Skip-Frame" Strategy 26

4.10 A SAN Model for the System Using "Skip-Frame" Strategy . . . 26

4.11 Gate Specification for the System Using "Skip-Frame" Strategy . 27

4.12 Underlying Markov Chain for the "Skip-Frame" System 28

4.13 Computational Dependability as a Function of Coverage (I) . . . 29

4.14 Computational Dependability as a Function of Coverage (II) . . . 30

4.15 An Example System Having Two Data Sources 32

4.16 A SAN Model for the System Having Two Data Sources 32

4.17 Gate Specification for the System Having Two Data Sources . . . 33

4.18 Markov Chain for the System Having Two Data Sources 33

4.19 Failure Probability as a function of Iteration Rate 34

4.20 Reward for Different Maintenance Schedules ($e_0 = 0.0001$) 37

4.21 Reward for Different Maintenance Schedules ($e_0 = 0.002$) 38

4.22 Reward for Different Maintenance Schedules ($e_0 = 0.004$) 38

4.23 Reward for Different Maintenance Schedules ($e_0 = 0.006$) 39

4.24 Reward for Different Maintenance Schedules ($e_0 = 0.008$) 39

4.25 Reward Difference as a Function of Original Software Quality . . 40

6.1 A Hierarchical Approach to Model Construction and Solution . . 55

6.2 Recovery Blocks Operation 58

6.3 Dependability Sub-Model of RB 62

6.4 Performance Sub-Model of RB 64

6.5 Performability Model of RB 66

6.6 N-Version Programming Operation 70

6.7 Dependability Sub-Model of NVP 72

6.8 Performance Sub-Model of NVP 75

6.9 Performability Model of NVP 77

6.10 Comparisons of RB and NVP 81

6.11 Expected Reward for Non-Fault-Tolerant Software 83

6.12 Alternative NVP Operation 85

6.13 Dependability Sub-Model of Alternative NVP 87

6.14 Performance Sub-Model of Alternative NVP 93

6.15 Performability Model of Alternative NVP 96

6.16 Sensitivity to the Mean Execution Time of the Slowest Version . . 97

6.17 Comparisons of RB, NVP and Alternative NVP 98

6.18 Impact of Component Performance 99

6.19 Comparison for Uniform Distribution 100

6.20 Comparison for Open Loop Case 110

6.21 Comparison for Close Loop Case 110

7.1 Two Types of Modularity 115

List of Tables

6.1 Fault Types and Notations for RB 59

6.2 State Definitions for RB Dependability Model 60

6.3 Fault Types and Notations for NVP 71

6.4 State Definitions for NVP Dependability Model 73

6.5 Notations for Parameters 79

6.6 Assignments of Parameters for RB 80

6.7 Assignments of Parameters for NVP 80

6.8 Fault Types and Notations for new-NVP 87

6.9 State Definitions for new-NVP Dependability Model 88

6.10 Parameters of Uniform Distribution (RB) 101

6.11 Parameters of Uniform Distribution (NVP) 101

6.12 Parameters of Uniform Distribution (new-NVP) 101

6.13 Assignments of Parameters for RB 108

6.14 Assignments of Parameters for NVP 109

6.15 Assignments of Parameters for new-NVP 109

ACKNOWLEDGMENTS

I would like to thank my committee members, Professors Algirdas Avižienis, David A. Rennels, Jack W. Carlyle, Thomas Ferguson, Kirby A. Baker, and John F. Meyer for serving on my committee. Professor Meyer was especially instrumental in introducing me to performability and in guiding my research. I would like to thank Professor Ferguson for his help on many mathematical details.

I wish to express my sincere appreciation to Professor Algirdas Avižienis for his advice and encouragement that have contributed significantly to my graduate study and dissertation research.

Special thanks to Professor Richard Muntz whose advice became a key for the progress of my work.

Thanks also to Dr. Herbert Hecht for his steady support, encouragement and patience in the past three years.

Heartfelt thanks to my husband and best friend Kam Sing for his steady encouragement and support in many practical ways.

This research is partially supported by the Small Business Innovative Research (SBIR) Contract F33615-C-1468 from the U.S. Department of Defense. I want to acknowledge the support and interest of AAAF-3 of Wright Laboratories and Mr. Marc Pitarys, the technical monitor.

Vita

1984	B.S. (Computer Science), UCLA.
1985–1986	Research Assistant, Computer Science Department, UCLA.
1986	M.S. (Computer Science), UCLA.
1986–present	Research Engineer, SoHaR Incorporated, Beverly Hills, CA

Publications

Herbert Hecht, Myron Hecht and Ann T. Tai, "Software Certification Testing," *Proc. Ninth Annual Software Reliability Symposium*, Colorado Springs, Colorado, May 1991

Ann T. Tai, John F. Meyer and Herbert Hecht, "A performability model for real-time software," *Proc. First International Workshop on Performability Modeling of Computer and Communication Systems*, Enschede, Holland, Feb. 1991

Herbert Hecht, Ann T. Tai and John F. Meyer, *Avionics Software Performability*, prepared for USAF Wright Laboratories under contract F33615-90-C-1468, Jan. 1991

Wesley W. Chu, Andy Hwang, Herbert Hecht and Ann T. Tai, "Design Considerations of a Fault Tolerance Distributed Database System by Inference Technique," *Proc. International Conference on Databases, Parallel Architectures, and Their Applications*, Miami Beach, Mar. 1990

Ann T. Tai, Myron Hecht and Herbert Hecht, "A Testing Methodology for Critical Software," *Proc. of COMPSAC '87*, Tokyo, Japan, Oct. 1987

John P. J. Kelly, Algirdas Avizienis, Brad T. Ulery, Barbara J. Swain, Rung-Tsong Lyu, Ann T. Tai and Kam-Sing Tso, "Multi-Version Software Development," *Proc. of the Fifth IFAC Workshop, SAFECOMP'86*, Sarlat, France, Oct. 1986

A Study of the Application of Formal Specification Methods for Fault-Tolerant Software, Technical Report, CSD-880100, Computer Science Department, University of California, Los Angeles, June 1986

ABSTRACT OF THE DISSERTATION

Performability Concepts and Modeling Techniques for Real-Time Software

by

Ann Tsu-Ann Tai
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Algirdas Avižienis, Chair

To evaluate the operational properties of real-time software, performance and dependability must be considered simultaneously. Accordingly, the use of a unified performance-dependability measure for this purpose is highly desirable. A generic performability model is developed to incorporate such a measure. The model consists of a representation of the total system, together with the performance variable V. The total system comprises the real-time program (object system) in question and its operational environment. The model emphasizes the representation of interactions between performance attributes and dependability attributes and, moreover, captures the behavior of the software in its operational environment.

Real-time software systems may exhibit both gracefully and non-gracefully (the case in which an undetected error causes catastrophic failure) degradable performance. We employ a binary (or multi-nary) classification of operational integrity to govern the formulation for the unified evaluation. In this manner, distinctive yet coherent formulations of the performability measures can be provided across the boundaries of the classes. Accordingly, a reward structure is developed comprising the "top-down" and "bottom-up" (reward-based) capability functions. The former is employed to classify the software behavior in question using the concept of service threshold; whereas the latter realizes the evaluation of the degradable performance for the appropriate service class(es). The performance variable V is then formulated via the reward structure to summarize program performance over a mission.

The framework has been applied to the assessment of software fault tolerance techniques, namely, Recovery Blocks and N-Version Programming. A hierarchical approach is employed for model construction and solution. Performability modeling permits us to investigate the relationship between the cost of performance and the effectiveness of fault tolerance techniques. The unified measure provides to us insight about how to optimize the use of computational redundancy toward a performability objective in an operational context.

CHAPTER 1

Introduction

1.1 Motivation

Separate performance and dependability models are inadequate for optimizing software design because they usually preclude:

1. the evaluation of system performance under fault conditions for systems which exhibit degradable service;
2. an explicit representation of the environmental effects on dependability; and
3. the analytical study of the performance penalty due to partial failure or error recovery.

A unified measure overcomes these difficulties and permits

1. the study of various interactions between performance and dependability attributes during real-time software operation;
2. the resolution of conflicting goals of performance and dependability given limited resources for software development process.

In the evaluation of real-time software, such as avionics applications, it is particularly important to consider performance and dependability simultaneously.

Mission related metrics of the impact of avionics system modifications that incorporate both performance and dependability, directly related to elements of software design, can provide the means for sound engineering tradeoffs [1]. Hence, a model integrating performance and dependability assessment and translating benefits from either field into the value of real-time software in user terms is highly desirable. This model is aimed at capturing the interactions between performance and dependability, such as 1) marginal performance with regard to real-time constraints, causing system degradation or failure; and 2) fault tolerance schemes that cause response time/throughput penalties during fault-free operation or recovery process, thus degrading system performance.

In summary, the objectives established for this research are:

1. To identify concepts of existing performability studies that can be carried over to real-time software performability modeling.
2. To establish a framework for evaluation of software performability which includes a total system model, and a performance variable to evaluate software service quality via a reward structure.
3. To evaluate the framework by applying it to the assessment of software fault tolerance techniques.

1.2 Basic Definitions and Associated Terminology

The basic definitions and associated terminology used in this dissertation follow the conceptual framework provided by [2] for expressing the attributes of what constitutes dependable computing.

Dependability is that property of a computer system that allows reliance to be justifiably placed on the service it delivers. The *service* delivered by a system is its behavior as it is perceived by its user(s).

A system *failure* occurs when the delivered service deviates from the specified service, where the service *specification* is an agreed description of the expected service. The failure occurred because the system was erroneous: an *error* is that part of the system state which is liable to lead to failure. The cause of an error is a *fault*. An error is therefore the manifestation of a fault in the system, and a failure is the effect of an error on the service.

The faults fall classically into two classes: physical faults and human-made faults. Human-made faults primarily are design faults. Design faults are committed either during the initial specification and design or during subsequent system modification or maintenance procedures. The major distinction between the dependability attributes of hardware and software is: software systems present only design faults because there is no component deterioration.

In the context of fault-tolerant system employing computational redundancy and design diversity, the results of individual system components are allowed to differ within a certain range. Similar results are defined to be two or more results that are within the range of variation that is allowed by the decision algorithm used in a fault-tolerant system. Accordingly, when two or more similar results are erroneous, they are called similar errors.

1.3 Organization of the Dissertation

Concepts of existing performability models are described in Section 2 of this dissertation. A framework for the software performability model is presented in Chapter 3, and this framework is translated into executable implementations (stochastic activity networks) in Chapter 4, where examples of the application of the framework and executable models are also shown. To refine the framework, a reward structure based on a composite approach is presented in Chapter 5. Applications to the assessment of software fault tolerance techniques using the methods of hierarchical model construction are described in Chapter 6. Concluding remarks and future work are presented in Chapter 7. Appendix A provides a glossary of notation. Basics of stochastic activity network (SAN) structure are introduced in Appendix B. Samples of C code that were developed for the examples of software performability analysis are shown in Appendix C. Finally, Appendix D provides the sample Mathematica programs and their output for performability evaluation of fault-tolerant software.

CHAPTER 2

Prior Work and its Relation to our Research

2.1 A Brief Retrospective

A general framework for model-based performability evaluation was introduced by Meyer [3]. It permits the definition, formulation, and evaluation of unified performance-dependability measures. An important measure when observing a system during a specified period of time is the amount of cumulative benefits (reward) rather than the performance level at some point of time. A performability measure defined as the probability distribution function of cumulative performance during a specified period of time was introduced in [4]. The performability modeling and evaluation methods were applied to SIFT, an experimental fault-tolerant computer system for an air transport mission [5]. Stochastic activity networks (SANs) were developed to support (model-based) performability evaluation of complex systems [6] [7] [8]. A performability evaluation tool based on SAN, called METASAN [1], was developed to assist in the construction and solution of performability models [9]. The need for unified measures has also been recognized by other researchers with various approaches. Beaudry developed the notation of the computation availability to measure the effectiveness of the system [10]. Castillo and Siewiorek studied the relationship between workload, perfor-

[1]METASAN is a registered trademark of the Industrial Technology Institute.

mance and reliability for digital computing systems [11]. Chou and Abraham developed performance-availability models of shared resource multiprocessors [12]. Arlat and Laprie discussed performance-related dependability evaluation for supercomputer systems [13]. Iyer, Donatiello and Heidelberger described a recursive technique for computing moments of the distribution of accumulated rewards for repairable systems in [14]. Goyal and Tantawi developed an algorithm to compute the performability distribution in a heterogeneous system [15]. They carried out the analysis in the time domain to obtain a closed-form solution of the performability distribution. Hsueh, Iyer and Trivedi developed a measurement-based performability model using real error data collected on a multi-processor system. [16]. They defined a reward function based on the service rate and the error rate in each state in order to estimate the performability of the system and to depict the cost of different failure types and recovery procedures. Smith, Trivedi and Ramesh developed an algorithm to invert the transform equation using numerical methods for computing performability distribution [17]. Recently, Couvillion *et al.* described *UltraSAN* which uses stochastic activity networks and embodies many strategies to keep model size small, thus speeding simulation and aiding analysis [18].

A comprehensive survey of performability research work and results during the past 15 years can be found in [19].

2.2 Application of Performability Concepts to Software

Since the late 1970's, work on the development and application of performability models in a computing context has been primarily oriented toward hardware systems. Very few studies have considered the relationships between performance and dependability in software. Gelenbe and Mitrani studied the effect of errors and recovery on the execution time of Algol-like programs [20]. Chimento and Trivedi analyzed the execution time distribution of block structured programs run on processors subject to failure and repair [21]. Hsueh and Iyer's measurement-based performability model considers both hardware and software errors and reflects the interaction between system components [22].

However, a representation more tailored to real-time software is needed. Significant differences from hardware performability models include:

1. Real-time software systems may exhibit both gracefully and non-gracefully (the case in which an undetected error causes catastrophic failure) degradable performance.

2. While hardware degradation can often be measured as a function of physical elements, such as the number of processors or memory modules, software degradation is usually an abstraction and is often indicated by reduced accuracy, functionality, etc.

3. Software failure is exclusively the consequence of the interaction between program quality and the dynamics of the environment (there is no component degradation).

4. There can be failure behavior correlations: 1) between software components because of their precedence relation, or 2) between iterative executions of a single program due to data dependencies.

5. Untreated, delayed response constitutes an important failure category.

Therefore, we have developed a performability modeling framework that is suited to the particular needs of software evaluation [23]. The model consists of a representation of the total system, together with the performance variable V that quantifies a program's value (worth) in a specified operational environment. The generic performability model incorporates such a measure and is structured in a manner that facilitates its low level representation as an executable graphical model, e.g., a *stochastic activity network*.

The model emphasizes the representation of interactions between performance attributes and dependability attributes and, moreover, captures the behavior of the software in its operational environment. An important benefit of this approach is that it permits the explicit modeling of factors that cause software failures, such as the probability of executing a particular software module in an environment characterized by data variability, resource constraints, scheduling discipline, and fault tolerance capabilities. Some of the difficulties faced by current software reliability modeling that utilizes broad statistical measures such as fault density and failure rate, may be overcome by the adoption of this model.

To account for all these factors in a model of a real piece of software, performability evaluation tools are required. Hence, the model is generally defined in a manner that is conducive to lower level representation in a form that can be constructed and solved by either analytical methods or by existing performability evaluation tools. Specifically, the use of stochastic activity networks is considered.

Real-time software architectures, especially those employing defensive programming or fault tolerance strategies, may exhibit both gracefully and non-gracefully degradable performance. Quantitative measures from a straight unified evaluation may not correctly represent the service quality of a software system. The problem can be resolved by employing a binary (or multi-nary) classification of operational integrity to govern the use of a reward-based capability function. In this manner, distinctive yet coherent formulations of the performability measures can be provided across class boundaries. Accordingly, a reward structure is developed comprising the "top-down" and "bottom-up" capability functions. The former is employed to classify the software behavior in question using the concept of service threshold; whereas the latter realizes the evaluation of the degradable performance for the appropriate service class(es). This reward structure permits operationally meaningful and mathematically realizable performability measures, and thus refines the framework.

CHAPTER 3

Framework of Software Performability Modeling

The software performability concepts presented here are adaptations and extensions of earlier work concerning effects of transient and permanent hardware faults on a system's ability to perform [4] [5]. A software performability metric is a function of both program attributes and the conditions of the operational environment. As mentioned in the opening section, a software performability model consists of a representation of the total system, together with the performance variable V.

Definition 1 A *software performability model* is a pair

$$SP = \langle\, TS,\ V\,\rangle,$$

where TS denotes the total system in question and V is the performance variable quantifying the benefits of TS from its operation.

The total system and the performance variable are described in the following sections. To realize the evaluation of the performance variable, a reward structure capable to provide a rather complete assessment of real-time software service quality is developed in Chapter 5.

3.1 Total System

Definition 2 A *total system* is a pair

$$TS = \langle\, P,\ E \,\rangle,$$

where P is the program submodel, and E is the environment submodel.

A total system thus represents the real-time program (object system) in question and its operational environment. The object program has both structural attributes, which are static, and behavioral attributes which are dynamic and environment dependent. The operational environment includes factors external to the object program that affect its behavior, e.g., input behavior, computing resources, and mission profile. These need to be accounted for since input behavior is the driving force for design fault manifestations, computing resource availability is the basis for successful and timely program execution, and the mission profile conveys the time-varying nature of environmental conditions. The general hierarchy of the total system is illustrated in Figure 3.1.

All of the attributes can be expressed in mathematical terms. For example, in the structural part of a P-model comprising M software modules, the precedence relations between software modules can be described by an M by M matrix $\mathbf{Q}$, that is,

$$\mathbf{Q} = [q_{ij}], \tag{3.1}$$

where q_{ij} represents the precedence relation between software modules i and j, i.e.,

$q_{ij} = 0$, if sequence $(i;j)$ does not exist,

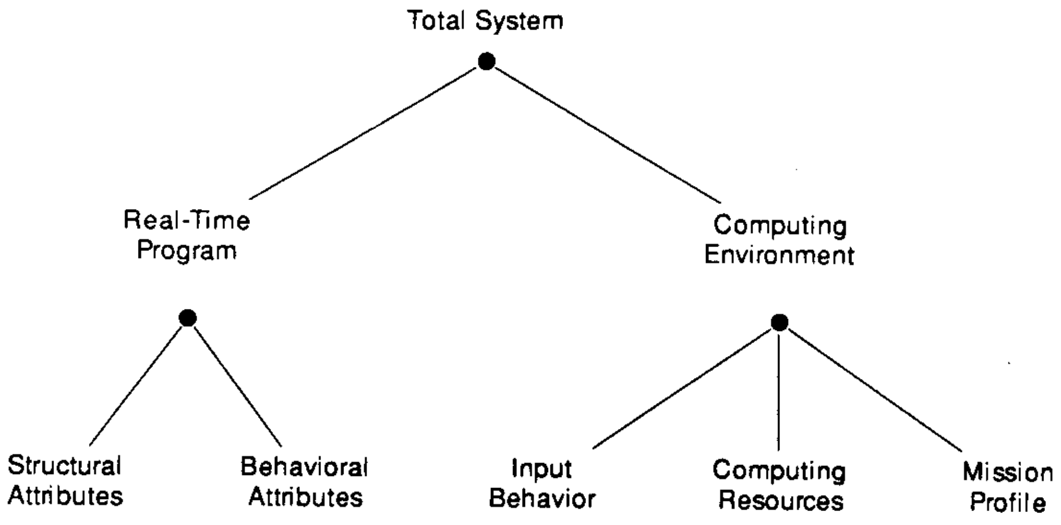


Figure 3.1: An Overview of the Total System

$q_{ij} = 1$, if sequence $(i;j)$ is unconditional,

$q_{ij} = \{predicate\}$, if sequence $(i;j)$ is conditional.

In the behavioral part of a P-model comprising M software modules, each of which is associated with K phases, the phased operational outcome of the software modules can be represented by an M by K matrix **OC**, that is,

$$\mathbf{OC} = [oc_{s,\phi}], \tag{3.2}$$

where $oc_{s,\phi}$ represents a success, degradation, or failure of the software module s (logic and timing) for the phase ϕ.

Components which make up the P and E-models can become degenerate under certain circumstances. For example, to model a software module whose behavior is phase independent, the "mission profile" component will consist of scalars describing a single phase mission instead of the vectors having elements indexed on mission phases. The extent of the elaboration of attributes in the P and

E-models (i.e., the expansion of the bottom level of the hierarchy of the total system shown in Figure 3.1) is determined by the characteristics of the system under evaluation and the focus of the modeling.

Conventional software dependability parameters such as failure intensity are not required in this software performability model. Instead, we emphasize the fact that real-time software failure behavior is a function of 1) the stochastic properties of the input behavior, 2) the probabilistic distribution of service requests and computing resource utilization, and 3) the capability of the fault-tolerance provisions. These parameters are represented in our model as follows. The stochastic properties of input behavior are represented in the E-model. Resource utilization and phase dependent service demands are attributes of the environment, hence system workload also is represented in the E-model. Software fault tolerance techniques usually involve one or more of the following: re-execution of code, branching to alternate modules, or concurrent execution, all of which can be expressed in terms of precedence relations, and are within the scope of the P-model. When performability is to be evaluated over a time interval that includes software maintenance, the effects of changes need to be modeled. This can be accomplished by varying the behavioral attributes and the resource utilization (to reflect changes in performance) and the input behavior (to reflect changes in fault locations).

3.2 Performance Variable

The performance variable V quantifies the performance of *TS* over a designated mission period T_ϕ (succession of mission phases), that is

$$V = \text{benefit from performance of } TS \text{ over } T_\phi.$$

The *performability* of *TS* is then given by the probability distribution function (PDF) of V. In certain applications, however, a less refined measure of V may suffice, e.g., the performability of *TS* is taken to be the expected (average) value of V. Evaluation of performability, however measured with respect to V, is based on a stochastic process which can be either an explicitly mathematical model or derived from a SAN graphical model of *TS*, along with a reward structure, which is usually implemented in a reward-based capability function $\tilde{\gamma}$ [4], defined on the state trajectories of that process. For a given trajectory u, the value of this capability function is the accumulation of reward impulses associated with operational outcomes (at the points of phase transition, or completion of an iteration of a program/sub-program). The outcome of a program execution is determined according to the performance and dependability requirements of the real-time system. The value of the reward impulse is based on the contribution of the outcome of program to the mission, which may be weighted by their phase-dependent criticality to the mission.

CHAPTER 4

SAN Realization of the Framework

SANs consist of the following primitives: activities, places, input gates, and output gates. Cases associated with activities permit the representation of uncertainties. Activities are of two types, timed and instantaneous. Elongated ovals represent timed activities and solid bars represent instantaneous activities. Places are depicted as circles. As with Petri nets, each place can hold a non-negative number of "tokens". The distribution of tokens in the places of the network at a given time constitutes the "marking" of the network at that time. Cases can be associated with both timed and instantaneous activities and are represented by small circles. The stochastic nature of the networks is realized by associating an activity time distribution function with each of the timed activities and a probability distribution with each set of cases. If exponential time distribution is associated with all activities, then the state marking behavior of the SAN model is defined by Markov processes, and can be described by the state-transition-rate diagram. Ultimately, the state occupancy probabilities may be obtained by standard Markovian solution techniques or simulation (as employed, for example, in METASAN). More details about SAN primitive elements and connection rules are introduced in Appendix B. SANs can naturally capture the characteristics of real-time software by exploiting the rich syntax of the generic SAN model. The methods for constructing the software performability model in SANs are illustrated in Section 4.1. The examples of computations and result analysis showing how SANs represent total system and support the performance variable are presented in Section 4.2.

4.1 Representation of the Total System

The total system model $TS = \langle P, E \rangle$ is constructed through interconnection of components of the P and E-models. Figure 4.1 depicts such an interconnection for the case where P consists of two modules in series.

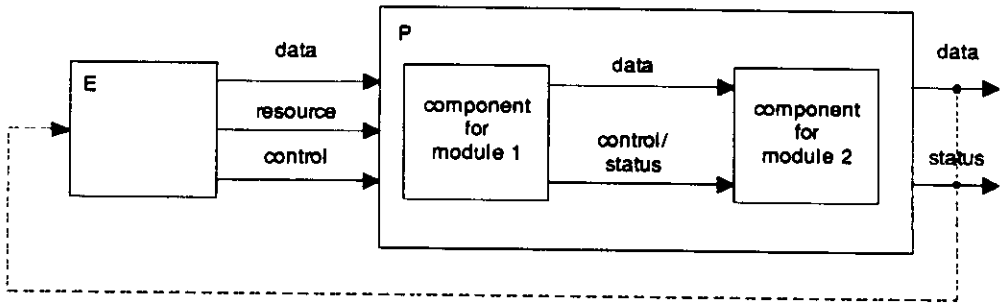


Figure 4.1: Example of a Total System

Between the components, the upper lines convey the data state and indicate data dependencies; the lower ones convey control and status information and indicate precedence relations. Between the E and P components there can also be computing resource status flow. The "feedback lines" from P to E represent the potential effects of software computation results and execution status on its environment. We illustrate the approach via an example. Suppose a real-time software system has a phase dependent scheduling policy (invocation rate is determined by mission phase). The interarrival time of external input data (from a sensor) is exponentially distributed. A fraction of the data in the input domain may trigger the manifestation of design faults in the software module. The data is transferred, at the time of invocation, to the software module's local data buffer. Data in the shared memory is always updated by the latest arrival. The time between invocations is deterministic and mission phase dependent. The time between mission phase transitions is also exponentially distributed. The nature of the input data determines whether or not the recipient software module will experience a failure. If an execution is not complete by the time of the next invocation, the prior execution is timed out. The next execution then starts with the latest input. The corresponding SAN model and its gate specification are shown in Figures 4.2 and 4.3, respectively.

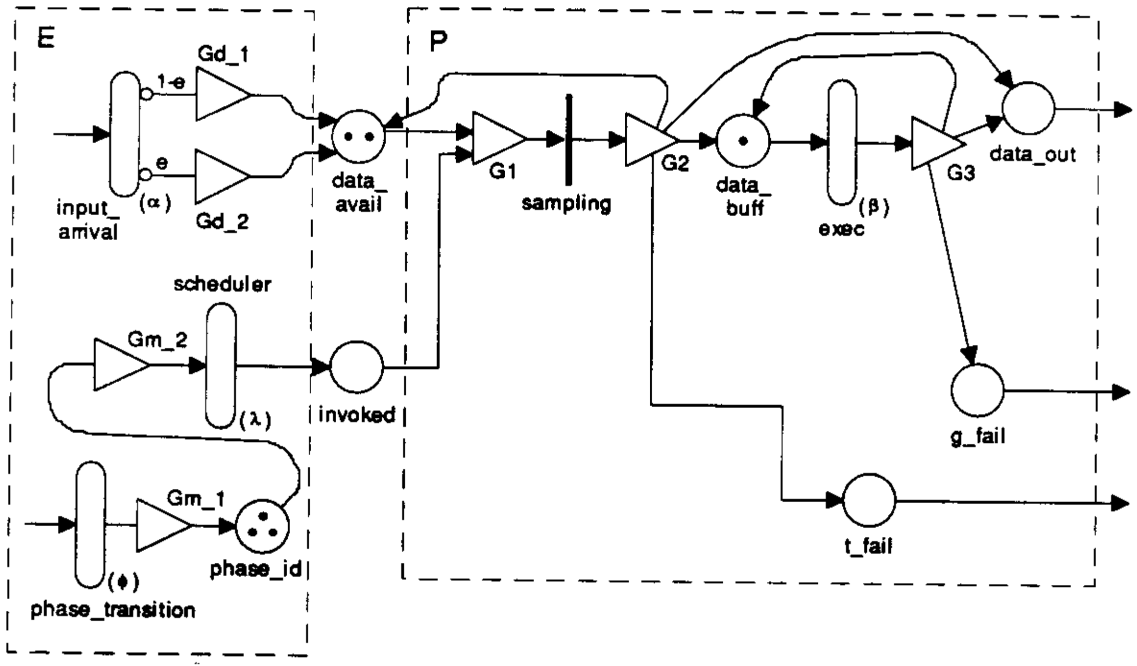


Figure 4.2: A SAN Model for Total System

The left part of Figure 4.2 is an E-model consisting of a data behavior com-

```
G1  [ inputs 1: data_avail;
             2: invoked;
    pred { (x1 && x2) }
    func { x2 = 0; } ]


G2  [ outputs 1: data_avail;
              2: data_buff;
              3: t_fail;
              4: data_out;
    func { if (!x4) { x3++; }
           x2 = x1; } ]


G3  [ outputs 1: g_fail;
              2: data_buff;
              3: data_out;
    func { x3 = x2; x1 += (x2 == 2); x2 = 0; } ]


Gd_1  [ output  data_avail;
      func { x1 = 1; } ]


Gd_2  [ output  data_avail;
      func { x1 = 2; } ]


Gm_1  [ output  phase_id;
      func { x1 = (x1 + 1) % (mi_length + 1); } ]


Gm_2  [ input phase_id;
      pred { (x1) }
      func { ; } ]
```

Figure 4.3: Gate Specification for Total System

ponent (the upper portion of the left box) and a mission profile component (the lower portion). A small percentage, ϵ, of data from the input domain may trigger program errors. The corresponding probabilistic information is conveyed by the cases associated with the activity "input arrival". Phase transition is specified by the gate "Gm_1" using a modulo function. The empty marking of the place "phase_id" represents the period between the missions, while each of the other markings represents the identification of a particular mission phase. The activity "phase_transition" has an exponentially distributed activity time, and the distribution can be phase dependent. The activity "scheduler" has a phase dependent and deterministic activity time and is always enabled, except between missions. The right part of Figure 3 is a P-model with its upper portion representing the execution information, and the lower representing the operational outcomes. At the time of invocation, input data should be "sampled" and stored in a buffer local to the P component. This is represented by an instantaneous activity "sampling". The data in the shared memory (represented by the marking of "data_avail") is allowed to vary during program execution, (e.g. from "error stimuli" back to "normal"). However, software module execution depends only on the data state at the beginning of execution, which is represented by the marking of "data_buff". If the execution activity does not complete by the time of the next invocation, the execution status will be altered. In other words, a timing failure is flagged by the addition of a token to the place "t_fail". The activity "exec" is then reactivated, which implies the restart of execution with the current input. Upon the completion of an execution, the marking of "g_fail" will flag a logic failure if the associated input data is an error stimuli. All of these "condition-action" rules are described in the gate specification. Via the interconnections among the SAN primitives, the total system model captures the interactions between the dynamics of P and E components in a natural way. Model parameters, such as the fraction of data that are error stimuli, can be obtained from software tests in which the inputs are random, but selected from data domains in proportion to their occurrence in an operational profile.

4.2 Example Analysis

This section presents the example models and analytical results demonstrating the capabilities of our software performability model. (Analytical solutions were implemented in C. Sample programs for the analysis are shown in Appendix C.)

4.2.1 Software Utilization, Success Criteria, and Expected Reward

Consider a sampled data system in which newly arriving information (synchronized with an invocation) can overwrite data in a buffer. For minimum lag between input and output it is desired to invoke the program operating on the sampled data very frequently, but as the time between invocations decreases there is increasing probability of overwriting the data buffer before the existing information has been processed. Assume the execution time is exponentially distributed with parameter (execution rate) β. The time between invocations is also exponentially distributed, with parameter α. If an invocation arrives before the previous execution is complete, the new invocation preempts the unfinished execution. A SAN model is shown in Figure 4.4. When a preemption (together with a data overwrite) occurs, one token is added to the place "timeout". When an execution is complete, the place is reset. The corresponding gate specification and underlying Markov chain are shown in Figures 4.5 and 4.6, respectively.

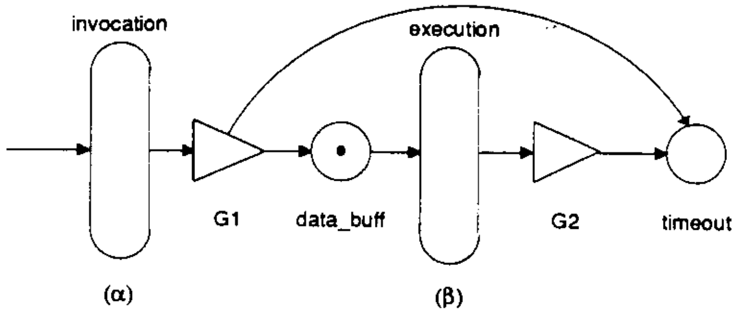


Figure 4.4: A SAN Model for a Sampled Data System

```
G1  [ outputs 1: data_buff;
              2: timeout;
    func { if (x1) {x2++;}
           else {x1 = 1;} ]

G2  [ output : timeout;
    func {x1 = 0;} ]
```

Figure 4.5: Gate Specification for the Sampled Data System

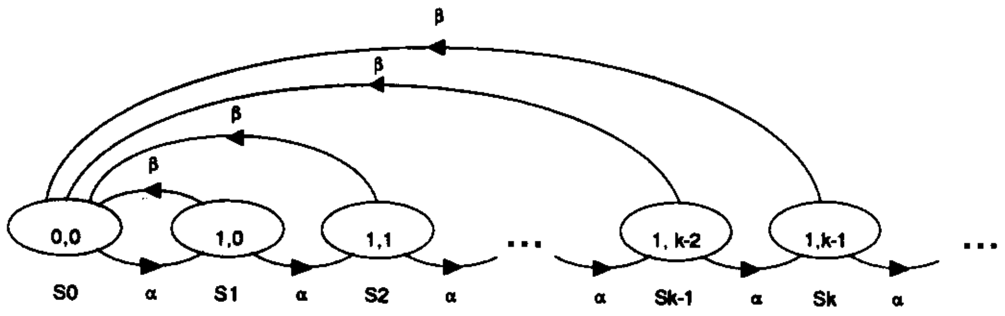


Figure 4.6: Underlying Markov Chain for the Sampled Data System

A relationship between software utilization and cumulative benefit is studied, specifically, the expected steady state reward per frame as a function of the invocation rate α (an indicator of software utilization). A positive reward impulse is assigned to an iteration upon its completion, while a negative reward impulse is assigned upon a time-out. Since invocations are Poisson arrivals, the number of iterations within a frame having a fixed length τ is a random variable. The expected reward accumulated in a frame (in the steady state) is the expectation of a random sum. Let w_s and w_f be the values of the positive and negative impulses, respectively. Further, let p_s and p_f be the steady state probabilities of iteration success and failure (time-out), respectively. Then the expected reward per frame (per interval of τ) is

$$E(V_\tau) = \sum_{k=0}^{\infty}\sum_{i=0}^{k}(w_s \cdot (k-i) + w_f \cdot (i))\binom{k}{i}(p_s^{k-i} \cdot p_f^{i})\frac{(\alpha\tau)^k}{k!}e^{-\alpha\tau}. \qquad (4.1)$$

Both the p_s and p_f can be obtained by solving the Markov model. The set of balance equations obtained from the Markov chain is:

$$\begin{aligned} \beta \cdot \sum_{k=1}^{\infty} P(S_k) - \alpha \cdot P(S_0) &= 0, \quad k = 0 \\ \alpha \cdot P(S_{k-1}) - (\alpha+\beta)P(S_k) &= 0, \quad k \geq 1 \end{aligned}$$

and hence we have

$$P(S_0) = \frac{\beta}{\alpha+\beta}, \qquad k = 0$$

and

$$P(S_k) = (\frac{\alpha}{\alpha+\beta})^k \cdot P(S_0), \qquad k \geq 1$$

then, $p_f = \sum_{k=2}^{\infty} P(S_k)$, and $p_s = 1 - p_f = P(S_0) + P(S_1)$.

As an example, we let $w_s = 1.0$ and $w_f = -5.0$. The results are shown in Figure 4.7. The expected reward increases as the invocation rate α increases up to a certain point. Beyond that point, further increases decrease the expected reward, because the larger number of missed iterations (execution time-outs) is not compensated by the gain from additional computations. The points of maximum reward show that a system with efficient code (indicated by a higher execution rate β) tolerates higher service demands (indicated by a higher invocation rate α) for maximum benefit. On the other hand, code with a long execution time (low β), must be restricted to lower rates to avoid excessive missing iterations.

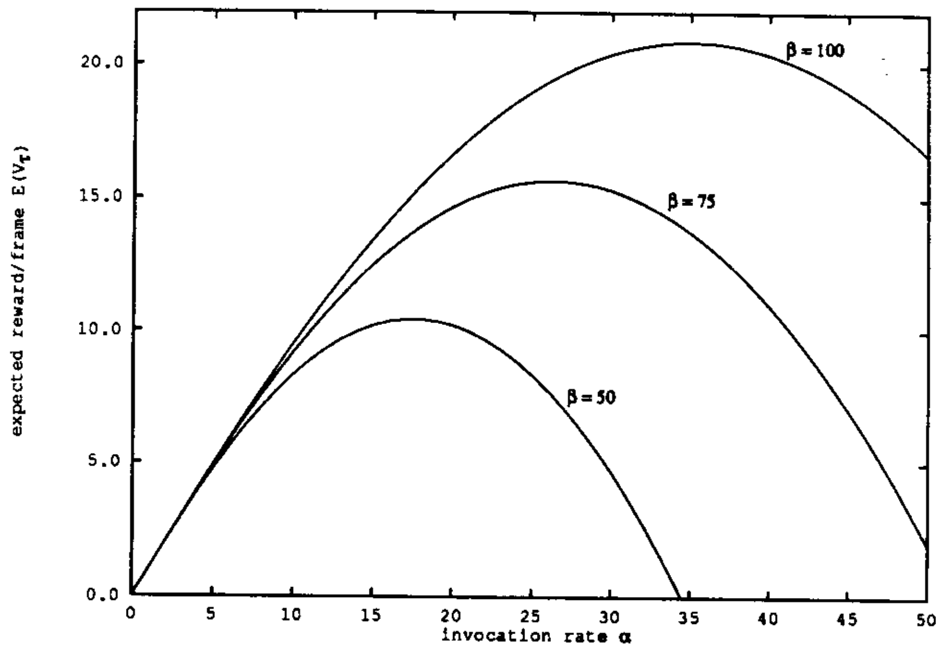


Figure 4.7: Expected Reward as a Function of Invocation Rate (I)

Now consider that the timing criterion is relaxed: the program is made resilient to one or more overwritten data. That is, an execution will not be counted as failed until its second, or third time-out. We refer to the maximum number of time-out tolerable as the "threshold" h. Accordingly, $p_f = \sum_{k=2+h}^{\infty} P(S_k)$, and

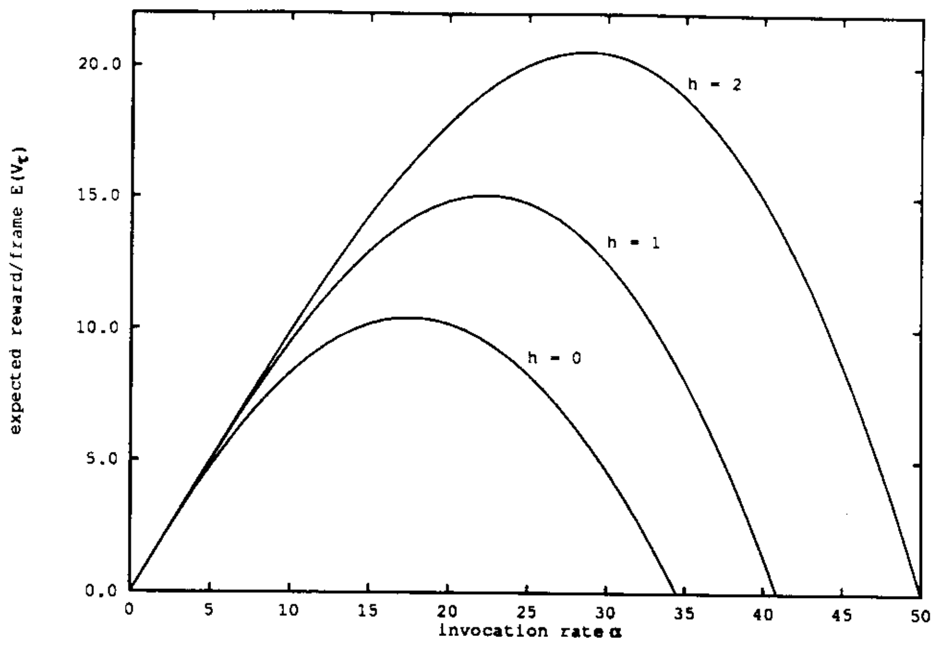


Figure 4.8: Expected Reward as a Function of Invocation Rate (II)

$p_s = 1 - p_f = P(S_0) + P(S_1) + \ldots + P(S_{1+h})$. However, as the timing criteria is relaxed, the latency caused by a timing failure is amplified by the increased time-out threshold. Accordingly, the negative reward for a discarded execution is increased by its original value multiplied by the time-out threshold. The results are illustrated in Figure 4.8. The points of maximum reward shift toward the right as the threshold for time-out is relaxed from zero to two. This shows that the system effectiveness can be improved if it is possible to relax the success criteria.

4.2.2 Coverage and Computational Dependability

This section presents numerical results from the performability modeling of a small-scale system shown in Figure 4.9. A preprocessor accepts data, invokes a software module (main process) and transfers the sampled data to it. The time between successive invocation/data transferring events is exponentially distributed with parameter α. The software module contains residual design faults, causing a fraction of the sensor input data from the preprocess to trigger failure of the main process. Erroneous results may be detected by an acceptance test in the main process. Upon the detection of an error, the main process applies the "skip-frame" strategy in order to achieve fault tolerance, i.e., a single missing iteration (detected erroneous computation) can be masked by using the result from the previous iteration. However, consecutive misses will cause failure. The execution time of the main process is exponentially distributed. Further, if an iteration is not completed by the time of the next invocation, it is considered a time-out condition and a new iteration begins using the current input. Consecutive time-outs also constitute a failure. The SAN representation of this system, the gate specification, and the corresponding Markov chain are shown in Figures 4.10, 4.11 and 4.12, respectively. In Figure 4.10, the left part is the E-model representing data behavior and service demands, while the right part is the P-model representing the software module's operational status and outcome. The cases associated with the activity "invocation/data_transfer" indicate the probability that an input is a potential error stimulus. The cases associated with the activity "exec" indicate the coverage of the fault tolerance mechanism — the probability of a successful error detection by the acceptance test. High coverage of error detection is usually achieved by sophisticated checks and assertions which may consume significant computation time [24]. Hence, the execution time of the main process is a direct function of coverage. Accordingly, it is assumed that the mean execution time ($\frac{1}{\beta}$) increases exponentially with coverage.

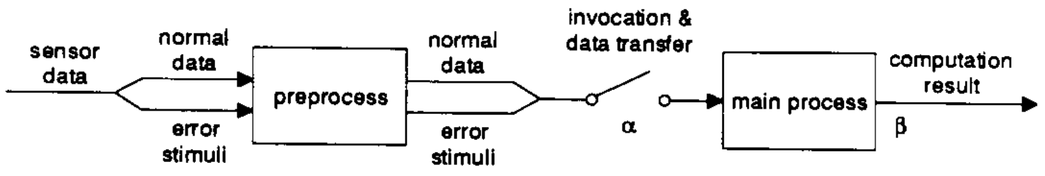


Figure 4.9: An Example System Using "Skip-Frame" Strategy

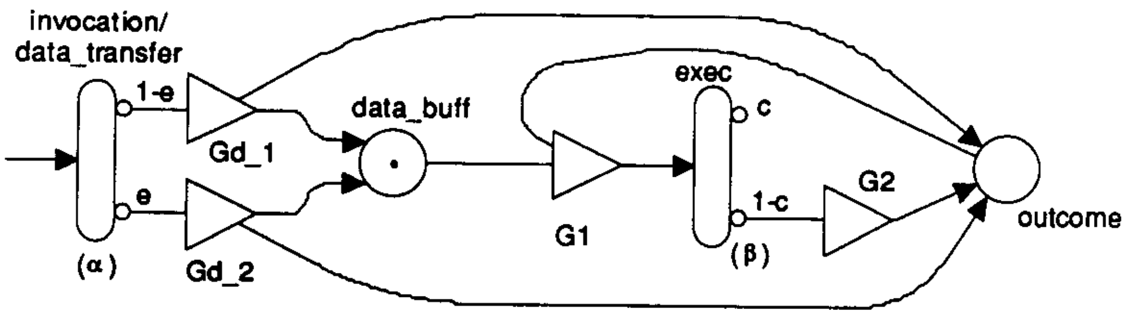


Figure 4.10: A SAN Model for the System Using "Skip-Frame" Strategy

```
Gd_1  [ outputs 1: data_buff;
                2: outcome;
      func { if (x1 >= 1) {
                if (x2 == 0) {x1 = 1; x2 = 2;}
                else { if (x2 == 2) {x1 = 0; x2 = 1; }}
             else {if (x2 == 2) {x1 = 1;}
                   else {x1 = 1; x2 = 0;}}
      }]

Gd_2  [ outputs 1: data_buff;
                2: outcome;
      func { if (x1 >= 1) {
                if (x2 == 0) {x1 = 2; x2 = 2;}
                else { if (x2 == 2) {x1 = 0; x2 = 1; }}
             else {if (x2 == 2) {x1 = 2;}
                   else {x1 = 2; x2 = 0;}}
      }]

G1  [ inputs 1: data_buff;
             2: outcome;
    pred { x1>=1 }
    func { if (x1 == 2) {
              if (x2 == 2) {x1 = 0; x2 = 1;}
              else {x1 = 0; x2 = 2;} }
           else {x1 = 0; x2 = 0;} ]

G2  [ output  outcome;
    func { if ( x1 == 2 ) { x1 = 1; } ]
```

Figure 4.11: Gate Specification for the System Using "Skip-Frame" Strategy

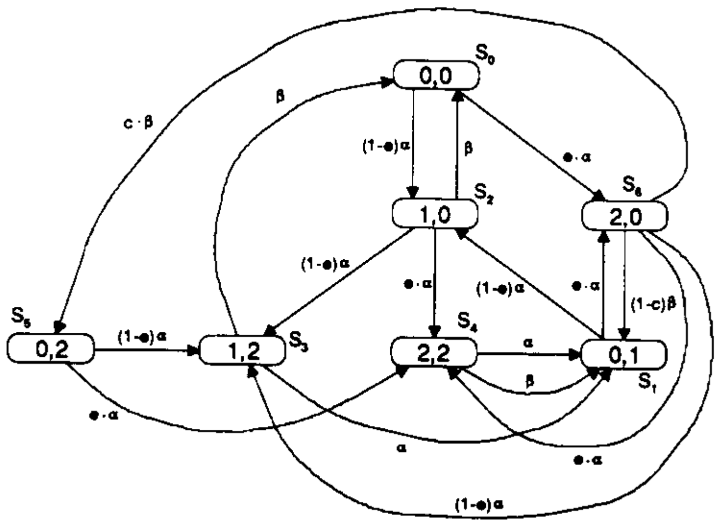


Figure 4.12: Underlying Markov Chain for the "Skip-Frame" System

Let p_s be the probability that an iteration of a software module will complete its task correctly and in a timely manner. We refer to this probability as *computational dependability*, a performance-related reliability measure of the type introduced by [10]. The dependencies between improvement of computational dependability and choice of the fault tolerance parameter — coverage, is investigated. In this example, $p_s = \frac{P(S_0)+P(S_6)}{P(S_0)+P(S_1)+P(S_6)}$. Figure 4.13 shows the computational dependability as a function of the coverage for the "skip-frame" strategy. The value of e increases from top to bottom. The bottom curve, with the greatest e, increases monotonically with decreasing slope. This shows that the highest dependability is achieved with perfect coverage (1.0) if the original failure probability (indicated by e) is high. On the other hand, a significant decrease in dependability due to increasing coverage values is noted for the upper curve. This indicates that the fault tolerance mechanism is likely to reduce computational dependability when the original failure probability is very low, because the performance penalty cannot be compensated by the reliability benefits. Each of the remaining curves reaches a maximum at less than perfect coverage. Beyond the maximum, the benefit of error masking due to coverage is outweighed by the performance penalty — the resulting number of time-outs. The maximum shifts toward the right (high coverage) as e increases, suggesting that it is beneficial to choose high coverage for software with high original failure probability. We also notice that the values of computational dependability, for different values of e, are quite close to one another in the perfect coverage region. This is due mainly to the fact that, if $c = 1.0$, all single errors are tolerated; hence, the resulting values of p_s are distinguished only by differences in the probability of consecutive errors. Since values considered for the e's are small, such differences are practically negligible.

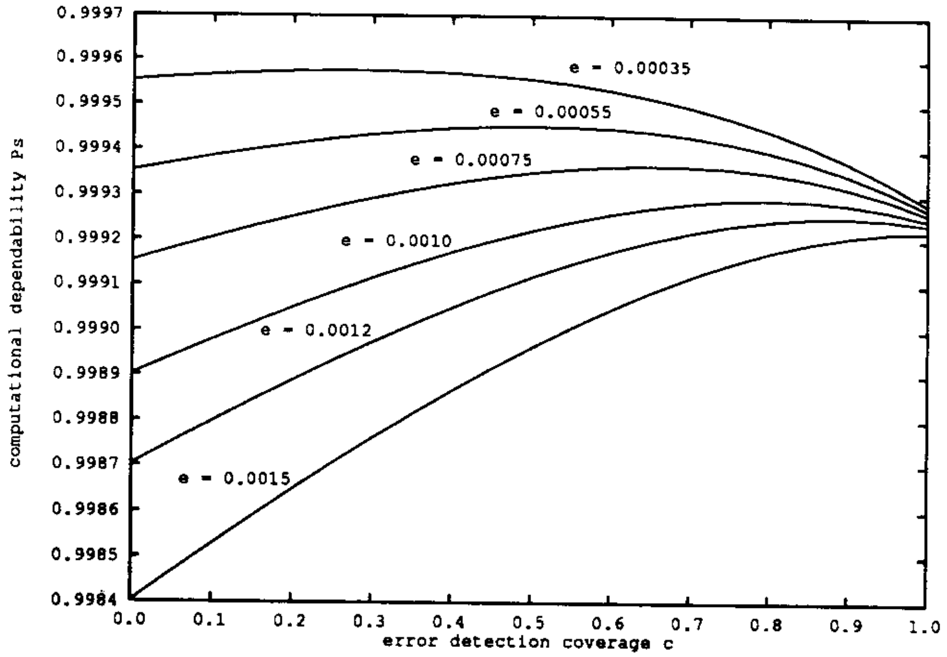


Figure 4.13: Computational Dependability as a Function of Coverage (I)

Figure 4.14 also shows the computation dependability as a function of the coverage of "skip-frame" strategy. Nevertheless, each of the curves has a different value of α — increasing from top to bottom. A larger α indicates higher service demands, higher time criticality and greater vulnerability to timing errors. The top curve with a small α is close to a straight line. This implies that the greatest benefit will be achieved at perfect coverage for the systems with low service demands. The curve next to the top is with a slightly larger α. It is also monotonic but its slope decreases as coverage increases, which implies that very higher coverage may not be cost-effective for the systems with fair workload. The lower curves in the figure are with larger α's which indicate heavier workload and greater vulnerability to timing errors. Each of the lower curves has a maximum. The maximum shifts toward the left (low coverage) as α increases, which implies that caution must be taken when choosing coverage for software with high service demands.

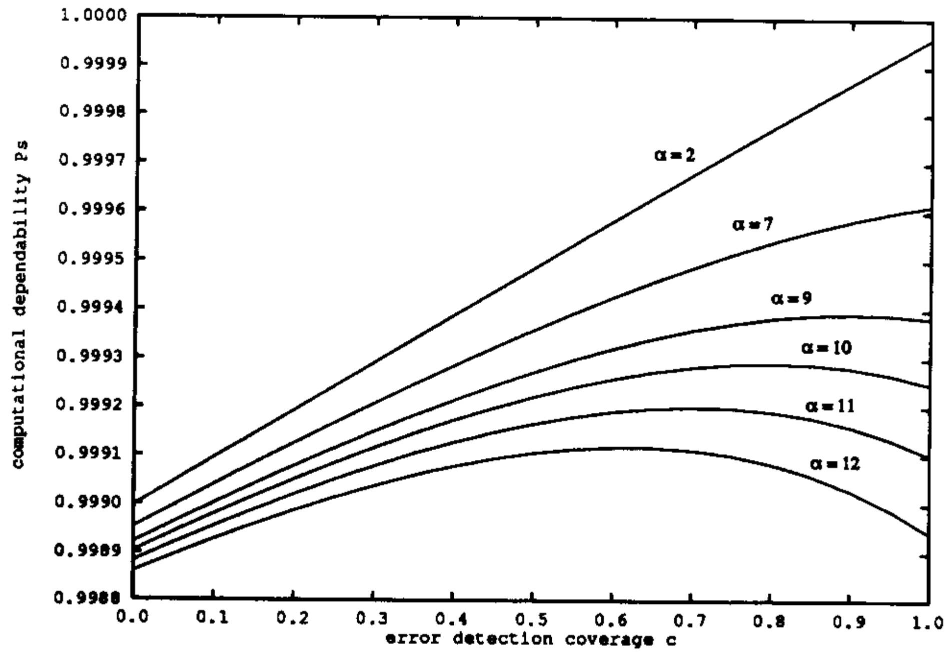


Figure 4.14: Computational Dependability as a Function of Coverage (II)

The analysis shows that the choice of fault tolerance coverage determines whether or not we can effectively achieve the desired computational dependability. The choice of suitable values for design factors can have considerable influence on performability, i.e., a program's ability to benefit its user. The optimum values are functions of both the operational environment and the program itself.

4.2.3 Workload and Software Failures

Consider another example system shown in Figure 4.15. The iterations of the module are looped: a new one starts upon the completion of the previous one. The module takes input from data source 1 at the beginning of iteration and from data source 2 near the end of iteration. A small percentage of data from source 1 are error stimuli, and data source 2 is subject to noise, which are Poisson arrivals with rate ν. The error stimuli from source 1 will result in fault manifestation while the noise from source 2 will completely disrupt the computation. In addition, it is assumed that error probability is a direct function of iteration rate (workload). The corresponding SAN diagram, gate specification and Markov chain are shown in Figures 4.16, 4.17 and 4.18, respectively. Failure probability (i.e., $P(S_1) + P(S_3)$) is computed as a function of workload (the iteration rate β). Figure 4.19 shows three curves with different noise rates. It is observed that the failure probability is high when the iteration rate is very low, decreases as the workload increases moderately, and increases when the workload becomes high. This phenomenon is due to the characteristics of the operational environment: when iteration rate is very low the chance that data source 2 accumulates noisy data before an iteration completion is high and program is vulnerable to the external errors; when iteration rate is high the chance of design fault manifestation is high and the program is vulnerable to the internal errors.

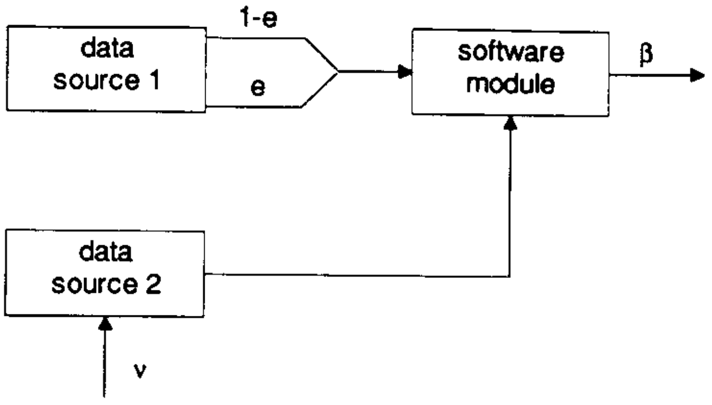


Figure 4.15: An Example System Having Two Data Sources

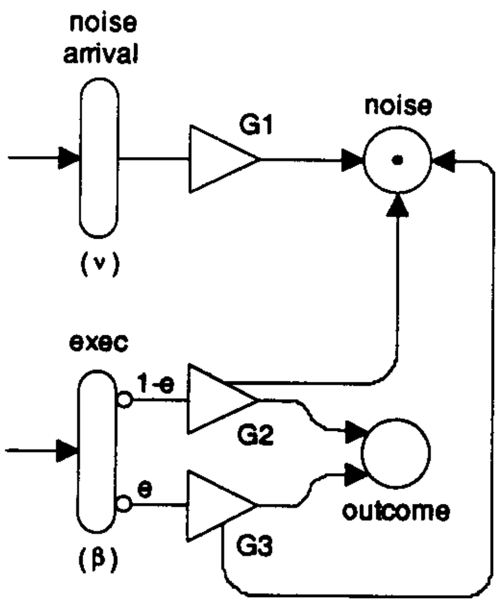


Figure 4.16: A SAN Model for the System Having Two Data Sources

```
G1  [ outputs : noise;
    func {x1 = 1;}]

G2  [ outputs 1: noise;
              2: outcome;
    func { if (x1) {x1 = 0; x2 = 1;}
              else {x2 = 0;}} ]

G3  [ outputs 1: noise;
              2: outcome;
    func {x1 = 0; x2 = 1;} ]
```

Figure 4.17: Gate Specification for the System Having Two Data Sources

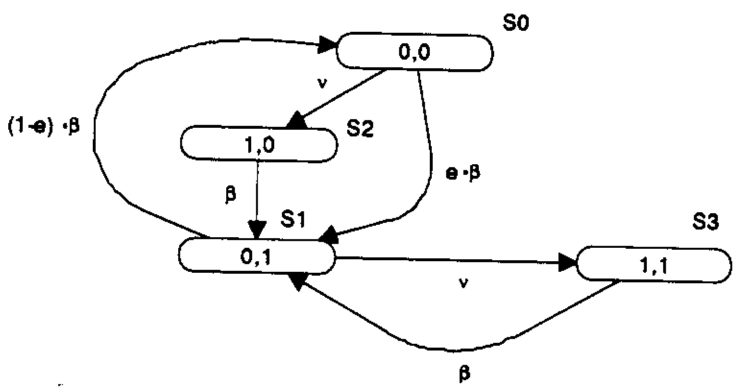


Figure 4.18: Markov Chain for the System Having Two Data Sources

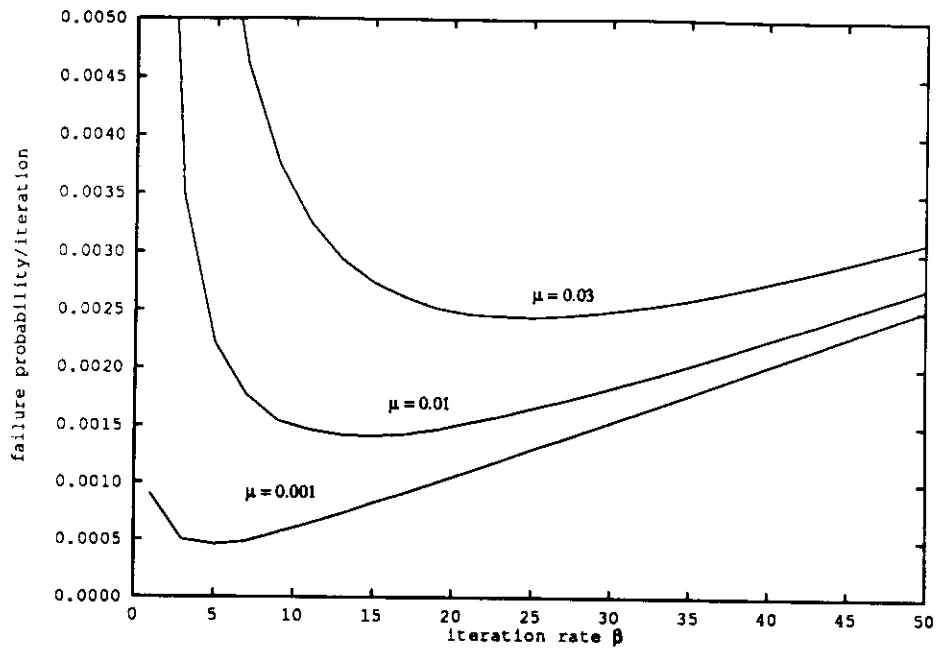


Figure 4.19: Failure Probability as a function of Iteration Rate

4.2.4 Maintenance Scheduling

This section presents an analysis of maintenance scheduling policies using performability prediction. This is an example of performability application in software development process. Conflicting demands in software maintenance activities arise from the following: from a performance perspective, perfective maintenance (performance or functional improvement) should be given priority; from a dependability perspective, corrective maintenance (fault removal) should be given priority. The factors for effective maintenance scheduling include:

1. Original program quality: characterized by the original error stimulus probability e_0 in the software performability model.

2. Maintenance quality:

- *Corrective maintenance quality* is characterized by the corrective factor e_r which is the percentage of the error stimulus probability remaining after the maintenance.
- *Perfective maintenance quality* is characterized by the perfective index $\frac{a}{b}$, in which a is the multiplication factor for the reward associated with a successful program execution to account for improvements due to the maintenance, and b is the multiplication factor for the error stimulus probability to account for the code added/modified during the maintenance.

The example system illustrated in Figure 4.9 is used for this analysis. The maintenance parameters are set as follows:

$$e_r = 0.2,\ a = 1.2,\ b = 1.2.$$

Assume that the maintenance activity for the software module starts at the conclusion of the first mission and is repeated for the two subsequent missions. Two possible schedules are 1) corrective-perfective-perfective, i.e., *C-P-P* and 2) perfective-perfective-corrective, i.e., *P-P-C*. We compare them by computing the cumulative reward over the four missions (the latter three reflecting the effects of the maintenance) and investigating the difference. Model parameters may change between missions due to maintenance (the changes are determined by the maintenance quality factors e_r, a and b), but model parameters stay unchanged within a mission. The expected cumulative reward for J missions is formulated as

$$E(\Gamma) = \sum_{i=1}^{J} E(\gamma_i),$$

where

$$E(\gamma_i) = E(\sum_{\theta=1}^{N} \gamma_\theta),$$

in which N is the number of frames in a mission and is assumed to be independent of γ_θ (the reward accumulated in a frame). We also assume that the mean lengths of missions are identical. Then,

$$E(\gamma_i) = \overline{N} \cdot E(\gamma_\theta).$$

If a frame has a fixed length of τ, then $E(\gamma_\theta)$ can be expressed as $E(V_\tau)$, and can be evaluated by Eq. (4.1). The procedure to compute the cumulative reward is as follows.

Step–1 Update model parameters according to the type of maintenance.

Step–2 Solve the Markov model for p_s and p_f.

Step–3 Compute the expected reward.

Step–4 Add the mission reward to the cumulative sum.

Step–5 If the computation is not complete for all the missions, go to Step-1.

Figures 4.20 to 4.24 illustrate the cumulative reward from the two maintenance schedules described above. The pairs of curves have different values of e_0, in an increasing order, from Figure 4.20 to Figure 4.24. Figure 4.20 shows that schedule *P-P-C* is more favorable over *C-P-P*, when e_0 is low. The reward difference decreases when e_0 increases, as shown in Figure 4.21. The difference becomes negligible when $e_0 = 0.004$, as shown in Figure 4.22. Further increase of e_0 results in sign change of the reward difference as shown in Figure 4.23, which implies that *C-P-P* becomes favorable when the original quality (from a dependability perspective) is fairly low. As e_0 further increases, the benefit of *C-P-P* becomes more obvious (Figure 4.24). We have also evaluated the cumulative reward for the schedule *P-C-P*. Each of the *P-C-P* curves falls between the corresponding *C-P-P* and *P-P-C* curves, which is a reasonable result. Figure 4.25 summarizes the interesting points from the preceding ten figures, the values of the reward differences fall on a straight line. The zero crossing provides a criterion in terms of original software quality (indicated by e_0) for scheduling decision. That is, if the original software quality is sufficiently high, new features can be introduced before corrective maintenance. Otherwise, corrective maintenance should take place before accepting new features. This demonstrates how even simple performability models can yield valuable insights into software maintenance strategies.

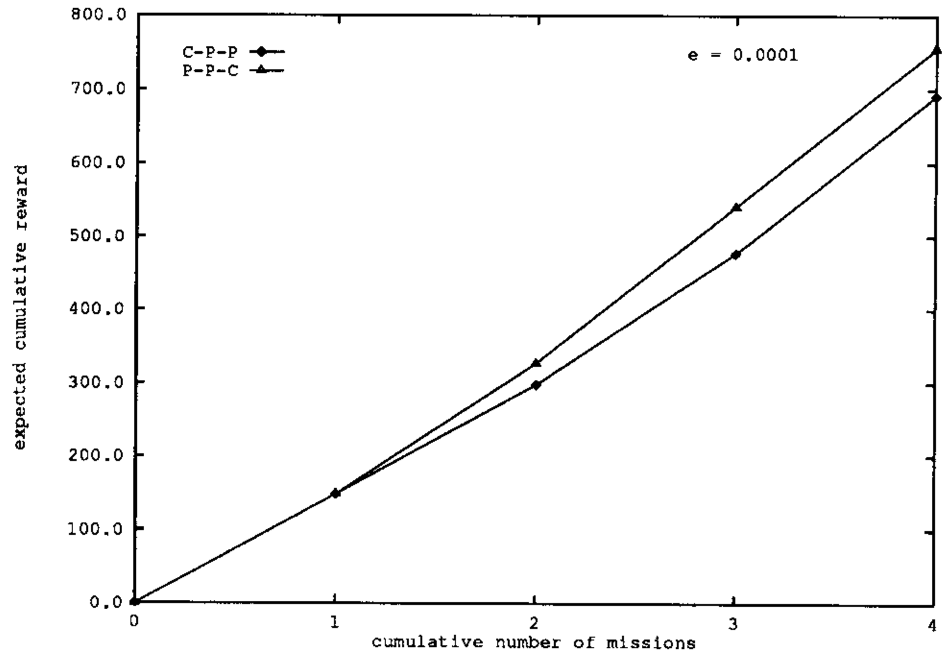


Figure 4.20: Reward for Different Maintenance Schedules ($e_0 = 0.0001$)

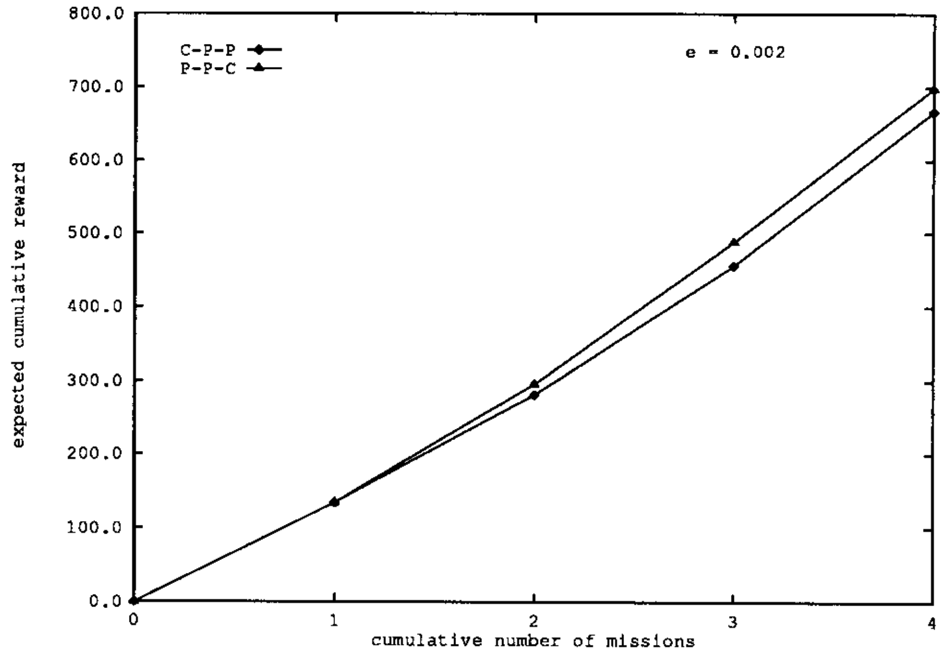


Figure 4.21: Reward for Different Maintenance Schedules ($e_0 = 0.002$)

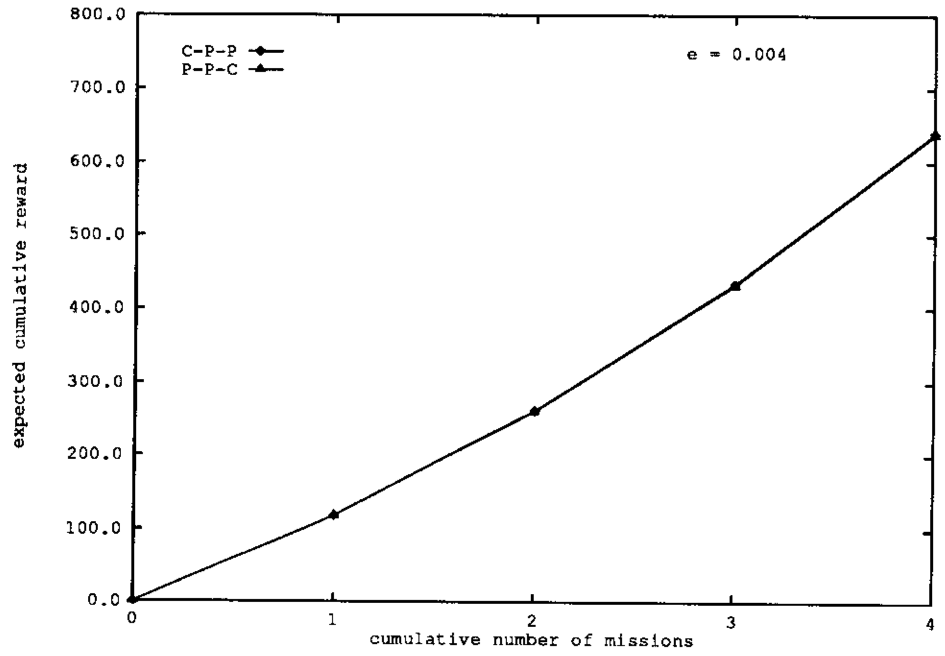


Figure 4.22: Reward for Different Maintenance Schedules ($e_0 = 0.004$)

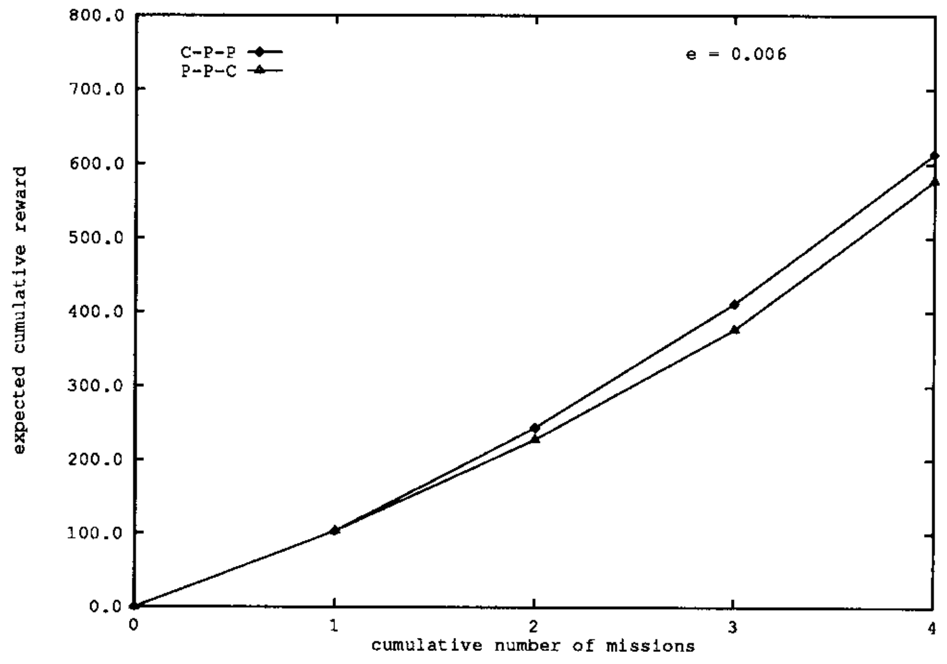


Figure 4.23: Reward for Different Maintenance Schedules ($e_0 = 0.006$)

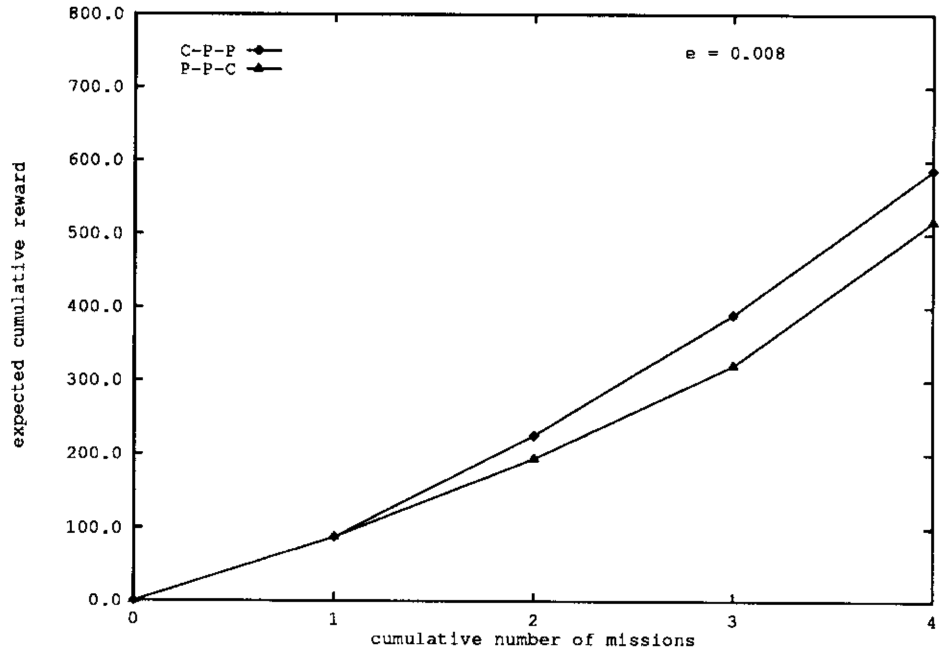


Figure 4.24: Reward for Different Maintenance Schedules ($e_0 = 0.008$)

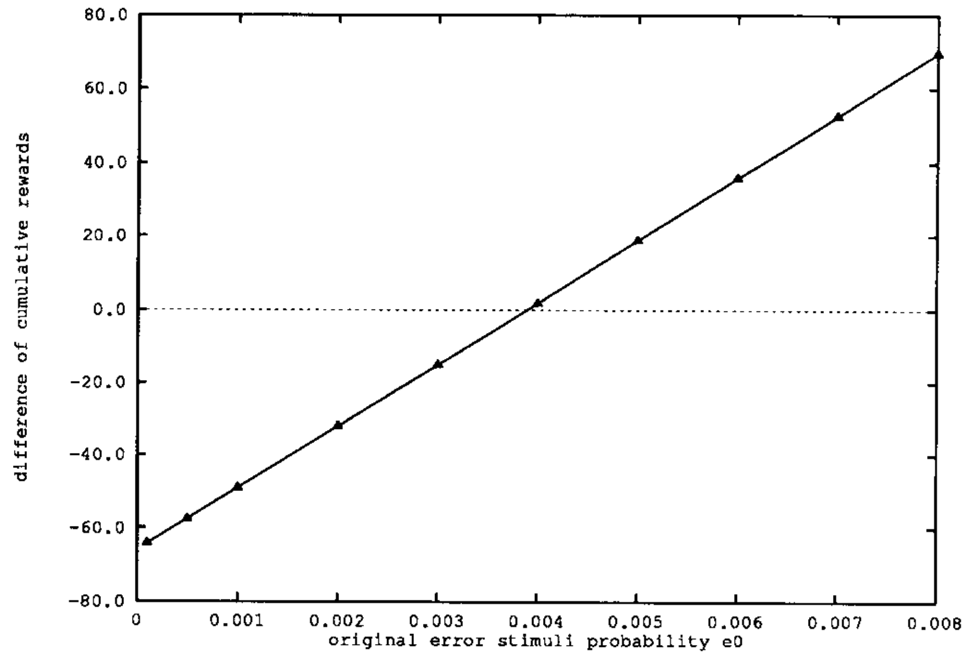


Figure 4.25: Reward Difference as a Function of Original Software Quality

CHAPTER 5

Refinement of the Framework: a Reward Structure

While simple and small scale problems, as presented in the previous chapter, can be evaluated in an *ad-hoc* manner, performability measures for more sophisticated real-time software applications need a rather formal and refined approach. Accordingly, we have developed a reward structure which enables a meaningful interpretation of the performance variable and makes its evaluation computationally practicable.

5.1 Difficulties in Performance Variable Formulation

In general, there are two approaches to support the definition and evaluation of a performance variable. The first approach is via the notion of a capability function which maps the state trajectory of the base model representing system behavior into user-oriented accomplishment levels [4]; accordingly, the performance variable associates with a countable and finite set of accomplishment levels. The second is through a reward structure which is a reward function associating reward rate or impulse with state occupancies or state transitions, respectively, in a base model [19], [25]; accordingly, the performance variable can assume a continuum of values. Both of the approaches support the evaluations of the system's ability to perform. Since the former emphasizes the user's point view, we call it a "top-down" capability function; and since the latter directly associates reward with the system behavior (usually a stochastic process), we call it a reward-based or "bottom-up" capability function.

In the performance variable formulation for real-time software, we face difficulties due to the following facts:

1. Due to data/state dependencies in the computation sequence (e.g., closed loops in avionics software), the effects of graceful service degradation may propagate over iterative program executions and affect the quality of subsequent computation.

2. Service loss (i.e., catastrophic failure caused by an undetected program error), or untreated excessive service degradation may negate the benefits accumulated from prior computation and alter the service quality from "proper" to "improper" with no warning (the case which corresponds to non-graceful degradation).

With the reward structure based on a direct "bottom-up" approach, such as the reward-based capability function, it is difficult to reflect the correlation between states and state transitions. Although a generic reward model allows reward values to be any real numbers [26], the magnitude of the negation entailed by an undesirable event in the sample path may be "prior states dependent." For example, the amount of the accumulated positive reward which needs to be erased upon a catastrophic failure depends on how many tasks succeeded before the failure. Thus the negative reward with the interpretation of a "penalty" or a "cost" of improper service is difficult to express analytically.

Furthermore, a reward structure based on the direct "bottom-up" approach is not able to explicitly take into account the service threshold. Neither the formulation of the performance variable nor its evaluation results can directly provide an indication of the boundary between "proper" and "improper" service. This lack of semantics makes the underlying unified measures less meaningful in practice. For example, system requirements usually define minimum performance and dependability requirements. Comparison of design alternatives based on quantitative unified measures are meaningful only when these minimum requirements are satisfied. It can be explained via a simple example. Consider a critical operational flight program which has a high iteration rate (high performance), but it generates an undetected erroneous output toward the end of the mission, causing a loss of the aircraft. Next, consider a functionally equivalent program which has a lower iteration rate (lower performance). It runs normally throughout the mission. If the performance variable is defined on the basis of the number of tasks (iterations) which succeeded through the mission period, or based on a time-averaged "task completion rate," the former case may correspond to a better reward than the latter. Thus the direct use of a reward-based capability function in software performability evaluation may result in bias.

On the other hand, a top-down capability function such as the one used for the SIFT evaluation primarily supports discrete performability variables ranging over a countable and typically a finite set of accomplishment levels [5]. A capability function of this type provides an operationally meaningful interpretation of performability measures with respect to both "bottom-line" performability requirements and gracefully degradable service. However, due to the fact that low level, design oriented details are often suppressed by a high level, discrete, and non-quantitative performance variable, the evaluation results may not be sufficiently informative to support the investigation of various design alternatives or tradeoffs.

5.2 Evaluating Unified Measures via Distinction

To circumvent the difficulties in evaluations of software performability, we revisited the concepts of "separate measure" and "unified measure" in performability terminologies.

When evaluating a system, one generally seeks to relate and quantify aspects of what the system is and does with respect to what the system is required to be and do [2], [19]. In the context of computing systems and with respect to a specified user-oriented or system-oriented service, performance typically refers to "quality of service, provided the system is correct." Dependability is that property of a system which allows "reliance to be justifiably placed on the service it delivers." Such service is proper if it is delivered as specified; otherwise it is improper. Hence, separate performance and dependability evaluations are distinguished by regarding "performance" as "how well the system performs, provided it is correct" and regarding "dependability" as "the probability of performing successfully." This distinction makes the separate evaluation only a partial assessment of service quality and inadequate for systems where performance is gracefully degradable. Performability concepts and modeling techniques have been established for generating unified measures of performance and dependability for gracefully degradable systems, thus enabling a complete evaluation of service quality.

The objective of a system providing gracefully degradable service is to continue to benefit the user in the presence of errors. Accordingly, the definition of "proper" service for such a system is not restricted to the extent of "error-free." Instead, service is said proper if and only if it does not violate the service quality threshold (per its specification).

Based on these concepts and definitions, the evaluation of gracefully degradable service is naturally contingent upon proper system behavior throughout a designated time period. On the other hand, non-graceful degradation (service loss) corresponds to the event of improper system behavior.

The above observations suggest that we ought to realize the unified measures via distinction. That is, let the micro-level quantitative description of service quality be conditioned by the macro-level qualitative classification of the system behavior. This is not equivalent to a simple combination of individual assessments of system performance and dependability since:

1. the macro-level classification is based on the service threshold which can be defined by both performance and dependability criteria, and
2. the micro-level quantification measures performance degradation under error conditions.

Based on the notion of realizing unified measures via distinction, we defined a reward structure. The reward structure simultaneously exploits the concepts of the "top-down" and "bottom-up" capability functions. A "top-down" capability function is employed to classify the operational integrity and to guide the formulation of the unified evaluation. In this manner, distinctive yet coherent formulations of performability measures can be provided across the boundaries of the classes. As mentioned earlier, a measure of gracefully degradable service is naturally conditioned by the event that service quality is proper throughout a mission (or other designated utilization period). Accordingly, a reward-based capability function is applied to the "proper service" class to quantify the quality of the gracefully degradable service. Under the "improper service" category, measures could be implemented to indicate the penalty of service loss or the degree of safety impact. The details about the reward structure are illustrated in the following section.

5.3 A Reward Structure Using Composite Approach

In order to capture the dependencies in the program behavior without introducing intractable states, we emphasize on a collective view of the state trajectory. A collective view of the state trajectory denotes the collection of the "accounting records" of the software behavior through a mission. This allows us to assign reward values to software behavior on a local basis from a global perspective. A local basis refers to the individual events such as an iteration of a program. A global perspective assures that the dependencies between the events along a sample path are taken into account. For example, in an open loop application where the next state depends merely on external input, such that degradation in computational accuracy does not influence the quality of the subsequent computation, a collective vector can be defined on a set of random variables. Let (i_s, i_d, i_c) be a collective vector in which the coordinates denote the number of fully successful iterations, the number of detected and recovered erroneous iterations, and the number of undetected erroneous iterations, respectively. With a global perspective, the worth of the fully and partially successful iterations is accounted for only if the number of undetected erroneous iterations is zero. As another example, in a closed loop application where the current computation depends on the program state generated by the previous iteration, a collective vector can be defined on a set of random variables as follows:

$$(i_1, i_2, \ldots, i_N, i_c),$$

in which i_1, i_2, …, i_N and i_c denote the number of iterations before the first degradation, the number of iterations before the second degradation, …, the number of iterations before the N^{th} degradation, and the number of un-detected erroneous iterations, respectively. With a global perspective, reward values are associated with the software behavior by taking into account for the effects of degradation which propagate along the succession of iterations.

To capture the semantics (user point of view) of the software behavior and to effectively take into account the benefit negation due to improper service, a "top-down" capability function is employed to classify the software behavior. The formulation of the capability function can be simple because of its purpose: it takes a collective view of the state trajectory as the function argument and employs a service quality threshold to translate the state trajectory with a collective view into one of the performance level sets (classes). For example, performance levels can just comprise two sets, namely, A_{proper} and $A_{improper}$. A capability function can then be defined as follows.

$$\gamma(u_{\{t_0, t_0+t\}}) \in \begin{cases} A_{improper} & \text{if } u_{\{t_0,t_0+t\}} \text{ violates the service threshold} \\ A_{proper} & \text{otherwise} \end{cases}$$

where $u_{\{t_0,t_0+t\}}$ is the collective view of a state trajectory. The reward structure with the composite approach can then be defined as:

$$Y_{\{t_0,t_0+t\}} = \begin{cases} \tilde{\gamma}(u_{\{t_0,t_0+t\}}) & \text{if } \gamma(u_{\{t_0,t_0+t\}}) \in A_{proper} \\ \breve{\gamma}(u_{\{t_0,t_0+t\}}) & \text{otherwise} \end{cases}$$

where $\tilde{\gamma}(u_{\{t_0,t_0+t\}})$ is a reward-based capability function which evaluates the worth of proper software behavior, and $\breve{\gamma}(u_{\{t_0,t_0+t\}})$ is another reward-based capability function which assesses the penalty.

Consider a single-program and single-mission system (SPSM) in which the program does not communicate with any other program during a mission, and the effects of software behavior do not propagate across missions. Assume that the program is a closed loop one in which the effects of degradation propagate over iterations but do not amplify. Let $i^d_{\{t_0,t_0+t\}}$ be an indicator random variable which represents the number of program iterations between the d^{th} and $(d+1)^{th}$ degradations for a mission starting at t_0 and having length t. Let $I_{\{t_0,t_0+t\}}$ be a collective indicator vector consisting of the random variables $i^d_{\{t_0,t_0+t\}}$. And let R be a reward vector consisting of the r^d coordinates, each of which is the reward impulse associated with an iteration between the d^{th} and $(d+1)^{th}$ degradation. If we assume the penalty of service loss is independent of the state trajectory leading to a catastrophic failure, then $\breve{\gamma}(u_{\{t_0,t_0+t\}})$ becomes a constant c. Accordingly, we have the following reward structure:

$$Y_{\{t_0,t_0+t\}} = \begin{cases} R \cdot I_{\{t_0,t_0+t\}} & \text{if } \gamma(I_{\{t_0,t_0+t\}}) \in A_{proper} \\ c & \text{otherwise} \end{cases}$$

where $R \cdot I_{\{t_0,t_0+t\}} = \sum_{d=0}^{\hat{D}} r^d \cdot i^d_{\{t_0,t_0+t\}}$ ($\hat{D}$ is the maximum degree of degradation allowed by the service threshold), and $c \leq 0$.

5.4 Discussion

To describe the behavior of real-time software over a mission period, a suited performance variable $Y_{\{t_0,t_0+t\}}$ is defined which is a function of the amount of computational tasks the program accomplishes during the mission period t. When we assume that the execution time of a program follows a certain type of distribution, the value of $Y_{\{t_0,t_0+t\}}$ is usually unbounded. Moreover, the underlying exact or asymptotic distribution of $Y_{\{t_0,t_0+t\}}$ can be continuous (see Chapter 6 for examples). Accordingly, the performance variable can be associated with a continuum of values.

While the probability distribution function (PDF) of a performance variable is often difficult to evaluate by analytical methods [19], it is, in general, analytically attainable at the macro-level (e.g., the probability of a performance level class). A probabilistic measure at the macro level, such as the probability of improper service, may indicate that the likelihood of a service threshold violation exceeds the bottom-line performability requirement. In that scenario, it is more legitimate to identify the negative contribution from individual performance or dependability attributes of the design than to globally quantify its deficiencies. On the other hand, if the macro-level measures prove satisfactory, then the less refined measures at the micro-level, such as expected reward, will quantify the system's overall effectiveness and support design alternative selection and tradeoffs. In other words, the probabilistic measures at the macro level screen out systems (or designs) which violate the threshold. Thus meaningful quantitative comparisons or tradeoffs can be effectively performed by employing the weaker measures at the micro-level.

The stated reward structure enables the composite use of the stronger and weaker measures at different levels so that the evaluation becomes meaningful and practicable.

The use of a collective view of state trajectory can be extended to evaluations for more sophisticated systems, namely, single-program and multi-mission system (SPMM), multi-program and single mission system(MPSM), and multi-program and multi-mission system (MPMM). To account for spatial dependencies (dependencies among the programs with "producer-consumer" relationships), a collective view can again be applied to simplify the model construction and solution. That is, to aggregate the correlated programs into a group, and to define the random variables that give the collective view of a state trajectory based on the group.

CHAPTER 6

Applications to Fault-Tolerant Software

6.1 Background

Research efforts have been devoted to the modeling and analysis of software fault tolerance techniques. A majority of the literatures focuses on the two most documented approaches, namely, Recovery Blocks (RB) [27], [28], [29], and N-Version Programming (NVP) [30], [31], [32], [2], [33]. Three major objectives can be identified from these efforts:

1. modeling and analysis of the dependability measures.

2. modeling and analysis of the performance measures.

3. detailed analysis of the correlated design faults in diversified software.

In the first category, research efforts include [34], [35], [36], [37], [38], [39], [40], [41], [42], and [43]; in the second category, performance of RB and NVP schemes are evaluated in [36]; in the third category, representative work includes [44], [45], [46], [47], [48], [49], and [50].

Compared with the first and the third categories, much less research effort has been devoted to performance study of fault-tolerant software. The study presented here concerns a combination of the first and second categories. We emphasize the impact of the performance cost on the overall effectiveness of the fault tolerance techniques. We are motivated toward:

1. analyzing the contributions of dependability and performance attributes to the overall effectiveness of software fault tolerance techniques.
2. investigating the interactions between the dependability and performance attributes in fault-tolerant software.
3. assessing and comparing the different approaches to software fault tolerance.
4. identifying the implications of software dependability and performance engineering, from analytical results.
5. exploring practicable approaches to model construction and solution for realistic applications.

6.2 Assumptions

Software fault tolerance techniques evaluated in this research are Recovery Blocks (RB) and N-Version Programming (NVP). The types of program addressed here are for real-time applications. These programs are executed in loops. An iteration involves the execution of the fault-tolerant software components such as voting, acceptance tests, and alternative routines according to the control logic of a fault tolerance scheme. At the beginning of an iteration, the program accepts input, and it provides output at the end of the iteration. If the output of an iteration is determined to be erroneous, it will be suppressed and/or some default value will be provided, and the system may resume normal operations in the next iteration.

However, an undetected error will cause a catastrophic failure, i.e., it is not possible to have a successful or degraded iteration which benefits the user for the remainder of the mission. Further, each iteration is under a real-time constraint. A real-time constraint is a deadline on the responsiveness of the computational task. We assume that in a software fault tolerance scheme, there is a watchdog timer in the supervisory system to detect the violation of a real-time constraint. That is, if the execution of an iteration exceeds the deadline, it will be aborted by the supervisory system and the program state will be restored to start a new iteration. The treatment for such a detected timing error is as the same as a detected logical error.

6.3 Definition of Performance Variable

Since computer performance benchmarking normally uses the measure "kilo iterations" (per time unit or per time interval), we define the performance variable as the number of successful iterations of a program in a designated time period t, as perceived by the user. The performance variable is denoted as $M(t)$. A successful iteration is counted as one unit in $M(t)$. A degraded iteration is considered as a partially successful one, and is counted as a fraction, ranging over $[0, 1)$. An iteration resulting in catastrophic failure rules out all successful operations accomplished previously such that it forces $M(t)$ to zero. $M(t)$ reflects both the performance and dependability attributes of a program. This performance variable is supported by the reward structure defined in Chapter 5. We first evaluate the performance measures for an open loop application. Because there is no data dependency between iterations in an open loop program, normal operation resumed after a degradation is able to provide the same service quality as if no degradation had occurred. Let $X_0(t)$ denote the nominally successful iterations in t, independent of whether it benefits the user; also let $N(t)$ denote the number of cycles which generate undetected erroneous output in t. The service threshold is defined as $N(t) \geq 1$. Accordingly, we have the following capability functions:

$$\gamma(u_{\{t_0,t_0+t\}}) \in \begin{cases} A_{improper} & \text{if } N(t) \geq 1 \\ A_{proper} & \text{otherwise} \end{cases}$$

and

$$\tilde{\gamma}(u_{\{t_0,t_0+t\}}) = w_0 \cdot X_0(t)$$

where w_0 is the worth of a successful iteration provided that the service is proper. The value of w_0 is set to unity according to the definition of $M(t)$. An iteration which results in a detectable error is considered to have no contribution to the mission and thus it is not accounted for in the reward. Assume that a service threshold violation makes the entire mission worthless, and that the impact is independent of the failure behavior details; hence we set $\check{\gamma}(u_{\{t_0,t_0+t\}})$ to zero. Then, we have the following reward structure:

$$Y_{\{t_0,t_0+t\}} = M(t) = \begin{cases} \tilde{\gamma}(u_{\{t_0,t_0+t\}}) & \text{if } \gamma(u_{\{t_0,t_0+t\}}) \in A_{proper} \\ 0 & \text{otherwise} \end{cases}$$

Note that the random variables $M(t)$ and $X_0(t)$ are conceptually distinguishable in the sense that the former is a user-oriented figure of merit while the latter is a system-oriented stochastic process. Numerically, they also differ from each other in the event of a service quality threshold violation, in which $M(t)$ equals zero while $X_0(t)$ could still be positive.

6.4 A Hierarchical Approach to Model Construction and Solution

In order to consider both performance and dependability attributes of fault-tolerant software without introducing intractable states, we use methods of hierarchical model construction and solution. That is, we build the model by layers, start analysis from the lowest layer, and then pass the parameters calculated from the lower layer to the upper.

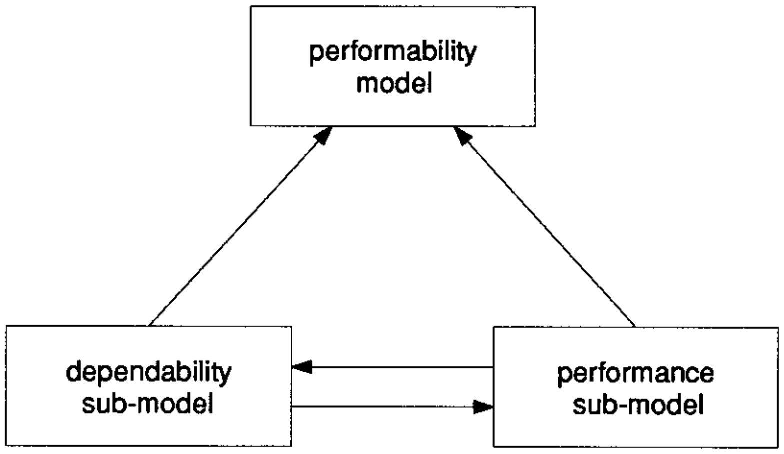


Figure 6.1: A Hierarchical Approach to Model Construction and Solution

A basic model is implemented in two layers, as shown in Figure 6.1. The lower layer consists of a dependability model and a performance model. The software failure behavior is represented by a discrete Markov chain in the dependability model. The dependability model is responsible for supplying the probabilities that a single iteration will succeed, degrade or fail. The performance model is a renewal process, in which each program iteration is represented by a renewal cycle. The performance model is responsible for supplying the mean and variance of the renewal cycle time (the time for a single iteration). It is assumed that the execution time of each individual component in the fault-tolerant software (e.g., a version, an alternative or a decision function) is independently and exponentially distributed. The mean and variance of the renewal time can be obtained using Coxian's method of stages and Laplace transforms [51]. The arrows between the lower layer blocks "dependability sub-model" and "performance sub-model" in Figure 6.1 indicate the information exchange between the two sub-models. That is, the dependencies between the two types of attributes are taken into account in the evaluation. Examples of the dependencies are: 1) the probability of real-time constraint violation as evaluated by the performance sub-model contributes to the probability of degradation (a dependability attribute), and 2) the probability of a detected error as evaluated by the dependability sub-model increases the probability of activating the use of the computational redundancies and thus has impact on mean program iteration time (a performance attribute). The information supplied by the lower layer forms the basis of a performability model at the upper layer. Since the mission time we are looking into (0.5 hour to 15 hours) is much greater than the renewal cycle time (in milliseconds), and the mean and variance are attainable, the renewal process has an asymptotic normal distribution [52]. That is, the performability model is a renewal process with an asymptotic normal distribution (defined on the parameters supplied by the performance sub-model), in which the outcome of each renewal cycle has a hyper-binomial distribution (defined by the parameters supplied by the dependability sub-model). Based on this performability model, a moment generating function of the performance variable can be derived, and thus the performability measures become attainable.

This is a logical way to construct and solve models for complex systems. The lower layer submodels are responsible for generating dependability and performance measures on the basis of an individual iteration. These results are then submitted to the upper layer model as parameters for the integrated evaluation. The semantics behind the model hierarchy are:

- The *lower layer* of the model represents the characteristics of a fault-tolerant system, namely, degradation and failure behavior, and operational disciplines.

- The *upper layer* of the model represents the effectiveness of the fault-tolerant system by taking into account the nature of the application and the user's view of service quality.

Therefore, as shown in the following sections, the performability models of different fault-tolerant software systems (RB, NVP, etc.) can share the upper layer when these object systems are evaluated for the same application (open or closed loop). On the other hand, the models for different applications can share the lower layer when they represent the same fault tolerance system.

6.5 RB Model

Figure 6.2 shows the operations of a RB scheme. The system has two functionally equivalent but diverse alternative programs, namely, the primary and the secondary. There is also an acceptance test which checks the correctness of the outputs of the alternatives. The supervisory system has a "watch dog timer" function. That is, a timer is set according to the deadline defined by the real-time system. The timer monitors the elapsed time from the beginning of an iteration.

If an iteration does not complete upon the expiration of the timer, it is aborted by the supervisory system and a new iteration starts.

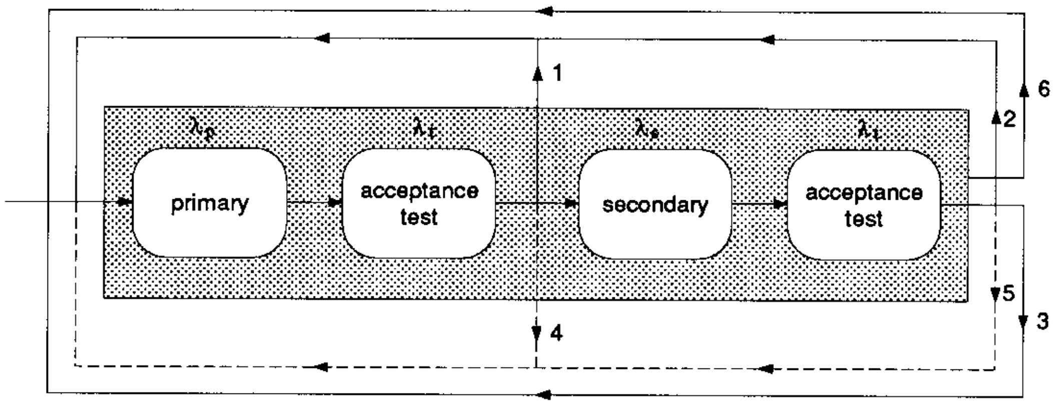


Figure 6.2: Recovery Blocks Operation

The system operates as follows. The primary (P) executes first, and the acceptance test (AT) runs upon the completion of the execution of P. If P computes correctly and AT accepts its results, the current iteration completes and the next starts (the case corresponding to path 1 in Figure 6.2). If AT rejects the results of P for any reason, the secondary (S) executes and AT subsequently checks the results of S. If S computes correctly and AT accepts the results, the iteration completes and the next starts (the case corresponding to path 2); if AT rejects the results of S for any reason, the result is suppressed and the next iteration starts (the case corresponding to path 3). Path 4 corresponds to the case in which P generates an erroneous result and AT subsequently accepts it; similarly, path 5 corresponds to the case in which S generates an erroneous result and AT subsequently accepts it, conditioned on the event that AT rejects P. From a stochastic process point of view, the iteration always resumes, independent of outcome type. From the user point of view, however, a catastrophic failure (caused by an undetected error) leads to loss of service. Therefore, paths 4 and 5 are represented by the dashed lines in Figure 6.2. Finally, path 6 corresponds to the case that the execution of an iteration exceeds its real-time deadline such that the iteration is aborted and the next cycle starts. Since this scenario may occur at any stage of an iteration, path 6 starts from the shaded rectangular instead of any component encapsulated inside.

6.5.1 Dependability Sub-Model

Using the concept of a total system, an operational software error is a consequence of the interactions between the error conditions in the object system and those in its operational environment. In that sense, the dependability submodel is a fault-manifestation model. Arlat *et al* developed rather complete fault classifications and fault-manifestation models for RB and NVP [41]. We adapt them in the lower layer as dependability submodels. The fault classification and notations for probabilities of fault manifestation for RB are illustrated in Table 6.1.

Table 6.1: Fault Types and Notations for RB

Fault Types	Probability of Manifestation
Related fault in P and S	q_{ps}
Related fault in P and AT (or P, S and AT)	q_{pt}
Related fault in S and AT	q_{st}
Independent fault in P or S	q_p or q_s
Independent fault in AT	q_t

The derivation of the fault manifestation model is based on the following assumptions:

1. No error compensation may take place within an alternative and the AT during an execution, i.e., an error is either detected and treated or leads to catastrophic failure;
2. The likelihood of singular behavior of the AT rejecting an acceptable result provided by P and subsequently accepting the result given by S is negligible.

The detailed model, a Markov transition state diagram based on the assumptions, is shown in Figure 6.3, in which

- $p_p = 1 - q_p - q_{pt} - q_{ps}$ is the probability that P computes correctly,
- $p_s = 1 - q_s - q_{st}$ is the probability that S computes correctly, conditioned by the event that no fault correlated to P is manifested, and
- $p_t = 1 - q_t$ is the probability that AT computes correctly, conditioned by the event that no fault correlated to P or S is manifested.

The definitions of the states are shown in Table 6.2.

Table 6.2: State Definitions for RB Dependability Model

States	Definition
I	initial state of an iteration
P	execution of P
$\{TP_i \mid i \in \{1,2,3,4\}\}$	execution of AT after P
$\{S_i \mid i \in \{1,2,3\}\}$	execution of S
$\{TS_i \mid i \in \{1,2,3,4\}\}$	execution of AT after S
B	benign failure
C	catastrophic failure

The partitioned states TP_i correspond to the various types of faults that may be manifested in P. The definitions are as follows:

TP_1 : no fault manifested in P.

TP_2 : manifestation of an independent fault in P.

TP_3 : manifestation of a correlated fault between P and S.

TP_4 : manifestation of a correlated fault between P and AT.

Subsequently, states S and TS are decomposed. After an independent error occurs in P, AT definitely leads to S_2 (a deterministic transition from TP_2) according to the definition of "independent error."[1] Manifestation of a correlated fault between P and S (state TP_3) corresponds to a detected error and leads through S_3 and TS_3 to state B. Manifestation of a correlated fault between P and AT (state TP_4) corresponds to an undetected error and leads to state C.

Let p_{cf} denote the probability of a catastrophic failure (the case corresponding to state C in Figure 6.3 or to paths 4 and 5 in Figure 6.2). And let p_{sd} denote the probability of benign failure (the case corresponding to state B in Figure 6.3 or to path 3 in Figure 6.2). Then from the Markov transition state diagram, we have:

$$\begin{aligned} p_{cf} &= p_p \cdot q_t \cdot q_{st} + q_p \cdot q_{st} + q_{pt} \\ &= (1 - q_p - q_{pt} - q_{ps}) \cdot q_t \cdot q_{st} + q_p \cdot q_{st} + q_{pt} \\ &= q_t \cdot q_{st}(1 - q_{ps}) + q_p \cdot q_{st}(1 - q_t) + q_{pt}(1 - q_t q_{st}) \\ &\approx q_t \cdot q_{st} + q_p \cdot q_{st}(1 - q_t) + q_{pt}. \end{aligned} \tag{6.1}$$

[1]Arlat *et al* stated that this unity transition is based on the assumption that no fault can be activated in AT after activation of an independent fault in P. We think that the definition of q_p determines this transition by logic, so that any further assumption in this regard will be redundant and contradictory.

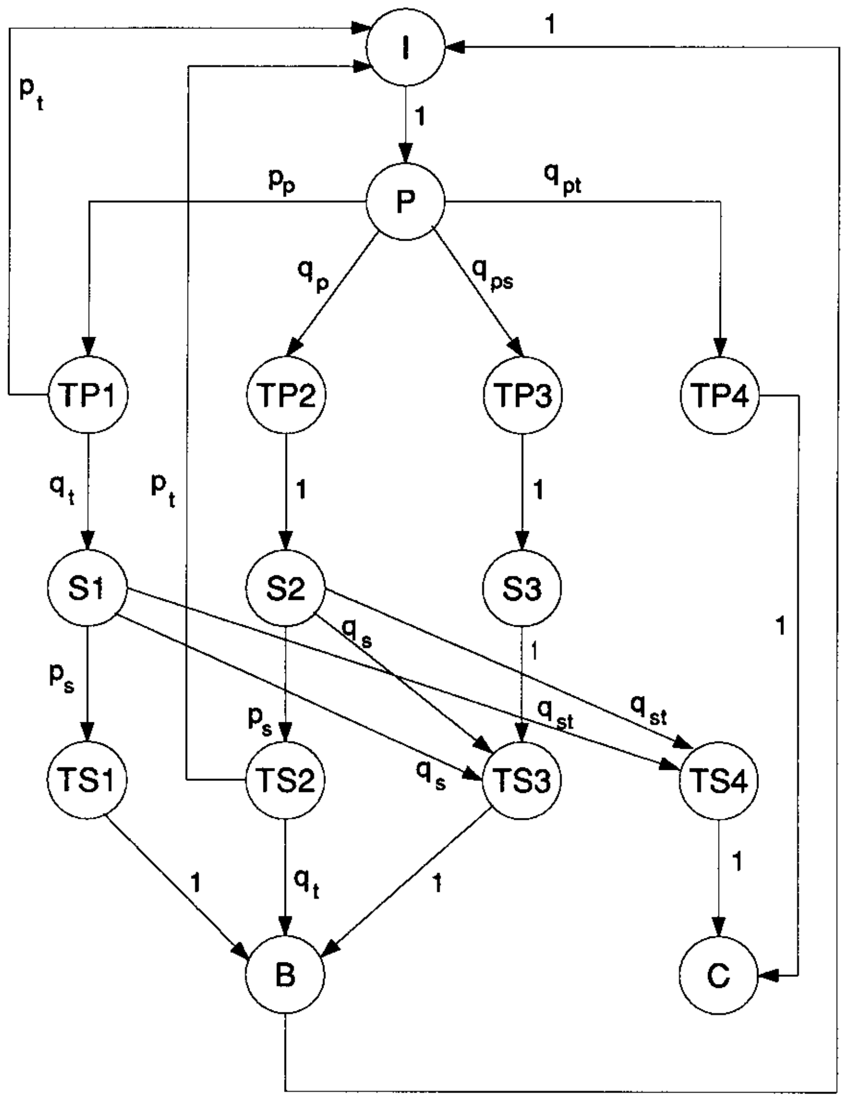


Figure 6.3: Dependability Sub-Model of RB

We notice that the third term in the above expression is the dominant term. And

$$\begin{aligned} p_{sd} &= p_p \cdot q_t \cdot (p_s + q_s) + q_p \cdot p_s \cdot q_t + q_p \cdot q_s + q_{ps} \\ &= (1 - q_p - q_{pt} - q_{ps}) \cdot q_t + (1 - q_{st}) q_p \cdot q_t (1 - q_s - q_{st}) + q_p \cdot q_s + q_{ps} \\ &= q_t \cdot (1 - q_{st})(1 - q_{pt} - q_{ps}) + q_p \cdot q_s (1 - q_t) + q_{ps} \\ &\approx q_t \cdot (1 - q_{pt} - q_{ps} - q_{st}) + q_p \cdot q_s (1 - q_t) + q_{ps}. \end{aligned} \tag{6.2}$$

In general, software fault tolerance schemes using redundancy are based on the assumption that the probability of correlated errors between components is much smaller than that of independent errors in a single component. Thus, the first term in the above expression (Eq. (6.2)) is the dominant one. In other words, the dependability of AT governs the benign failure probability. The other type of benign failure is the detected real-time deadline violation, which corresponds to path 6 in Figure 6.2. The analysis of benign failure of this type is presented in the next section.

6.5.2 Performance Sub-Model

We assume that the program runs in an open loop and let the number of total iterations in a designated time period t be denoted by a random variable $K(t)$. $K(t)$ is a renewal process in which each renewal cycle corresponds to a program iteration. Assume that the execution time of the primary, secondary and acceptance test are exponentially distributed, with parameters λ_p, λ_s, and λ_t, respectively. Figure 6.4 depicts this renewal process, which is derived from Figure 6.2. By definition, the random variable $K(t)$ is independent of outcome type, hence paths 1 and 4 in Figure 6.2 are aggregated into a single path marked p_1 in Figure 6.4; likewise, paths 2, 3 and 5 in Figure 6.2 are aggregated into a single path marked $(1 - p_1)$. The notation p_1 represents the probability that an iteration completes upon a successful run of P and AT or fails due to the similar errors between P and AT (caused by their correlated faults). From the dependability sub-model (Figure 6.3), we have,

$$p_1 = p_p \cdot p_t + q_{pt}.$$

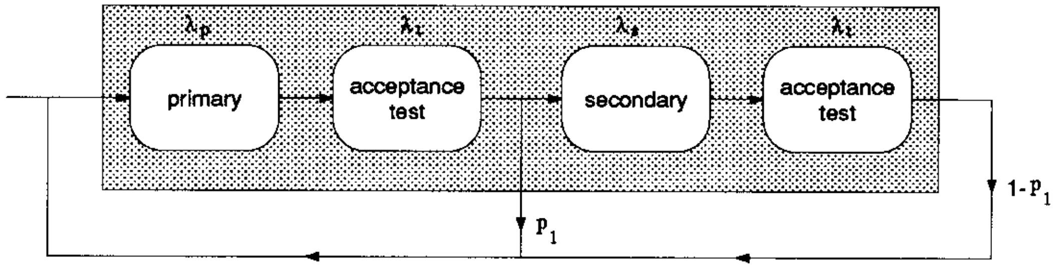


Figure 6.4: Performance Sub-Model of RB

Let the Laplace transform of the density functions (pdf's) of the execution time of the primary, secondary and acceptance test be denoted as $F_p^*(S)$, $F_s^*(S)$ and $F_t^*(S)$, respectively. Then the transform of the pdf of the renewal cycle time Y is:

$$F^*(S) = p_1 \cdot F_p^*(S)F_t^*(S) + (1 - p_1) \cdot F_p^*(S)F_s^*(S)(F_t^*(S))^2,$$

$$F^*(S) \Leftrightarrow f(y).$$

Through inverse Laplace transform, we obtain the density function of the renewal cycle time.

Due to a deadline τ (the real-time constraint), the iteration time Y follows the distribution defined by $f(y)$ if the iteration completes before the deadline; otherwise it stops at τ, which corresponds to path 6 in Figure 6.2 and can be represented by a unit impulse at τ scaled by the probability of deadline violation. Accordingly, the actual density function of Y is:

$$g(y) = \begin{cases} f(y) & Y < \tau \\ (1 - F(\tau)) \cdot \delta(y - \tau) & Y \geq \tau \end{cases}$$

where $\delta(y - \tau)$ is the unit impulse function and $F(\tau) = \int_0^\tau f(y)dy$. We let

$$p_{tt} = 1 - F(\tau) \tag{6.3}$$

which is the probability that an iteration violates the deadline. The probability p_{tt} together with the probability p_{sd} (Eq. (6.2)) constitute the probability of degradation, which is used by the upper layer model for performability evaluation. We can then compute the mean and variance of Y. That is,

$$\begin{aligned} E[Y] &= \int_0^\tau y\, f(y)dy + \int_\tau^\infty \tau\, f(y)dy \\ &= \int_0^\tau y\, f(y)dy + \tau \cdot (1 - F(\tau)). \end{aligned}$$

$$\begin{aligned} E[Y^2] &= \int_0^\tau y^2\, f(y)dy + \int_\tau^\infty \tau^2\, f(y)dy \\ &= \int_0^\tau y^2\, f(y)dy + \tau^2 \cdot (1 - F(\tau)). \end{aligned}$$

$$\mu = E[Y], \quad \text{and} \quad \sigma^2 = E[Y^2] - (E[Y])^2.$$

6.5.3 Performability Model

Figure 6.5 depicts the performability model at the upper layer. The shaded rectangular box corresponds to the shaded rectangular in Figure 6.4, which encapsulates the operational details of a renewal cycle (an iteration of the program).

As mentioned in Section 6.4, the renewal process $K(t)$ is approximately normally distributed. The two expressions marked in the box are the mean and variance of the asymptotic distribution (which are further explained late in this section). They are functions of μ and σ^2, the mean and variance of the renewal cycle time, supplied primarily by the performance sub-model. The outward arrows from the box denote the different outcomes of a renewal cycle, each of which is associated with a probability which is primarily supplied by the dependability sub-model. Contrasting Figure 6.2 to Figure 6.5, paths 1 and 2 in Figure 6.2 collapse to the path marked $(1 - p_{dg} - p_{cf})$ in Figure 6.5, paths 4 and 5 collapse to the path marked p_{cf}, and paths 3 and 6 collapse to the path marked p_{dg}. Intuitively speaking, the performance variable $M(t)$ corresponds to the number of cycles that are feedback from the upper path (successful iteration) in time t conditioned by the event that no cycle in $K(t)$ goes to the dashed path (undetected erroneous iteration).

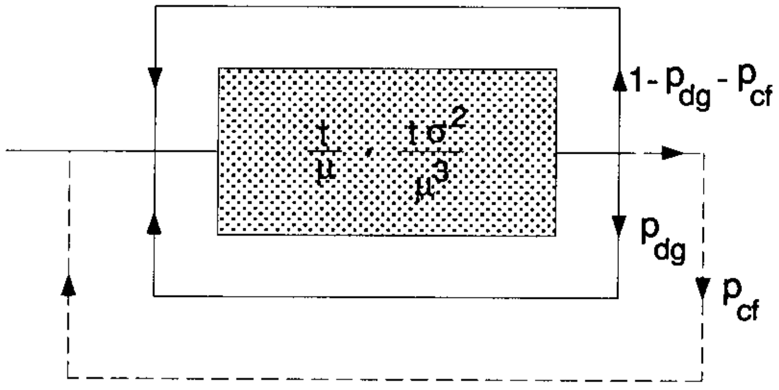


Figure 6.5: Performability Model of RB

Let $Z = e^s$, then the moment generating function of the performance variable $M(t)$ can be derived as follows.

$$E[Z^M] = \sum_{m=0}^{\infty} Z^m \cdot P(m \text{ in } t).$$

By rewriting the equation, we have

$$\begin{aligned} E[Z^M] &= Z^0 \cdot P(M(t)=0) + \sum_{m=1}^{\infty} Z^m \cdot P(M(t)=m) \\ &= P(M(t)=0) + \sum_{m=1}^{\infty} Z^m \cdot P(M(t)=m). \qquad (6.4) \end{aligned}$$

Let CF denote the event that there occurs one or more catastrophic cycles, and CF^c denote the absence of catastrophic failure, in the time period t. According to the definition of $M(t)$, we have

$$\begin{aligned} P(M(t)=0) &= P(M(t)=0 \text{ and } CF) + P(M(t)=0 \text{ and } CF^c) \\ &= P(CF) + P(M(t)=0 \text{ and } CF^c). \end{aligned}$$

We first compute $P(CF)$:

$$\begin{aligned} P(CF) &= \sum_{k=1}^{\infty} P(CF \mid K(t)=k) \cdot P(K(t)=k) \\ &= \sum_{k=1}^{\infty} (1-(1-p_{cf})^k) \cdot P(K(t)=k). \qquad (6.5) \end{aligned}$$

We then compute $P(M(t)=0 \text{ and } CF^c)$:

$$\begin{aligned} & P(M(t)=0 \text{ and } CF^c) \\ &= \sum_{k=0}^{\infty} P(M(t)=0 \text{ and } CF^c \mid K(t)=k) \cdot P(K(t)=k) \\ &= \sum_{k=0}^{\infty} {p_{dg}}^k \cdot P(K(t)=k). \qquad (6.6) \end{aligned}$$

where p_{dg} is the probability of degradation — the probability that a computational result is suppressed due to either a logic or timing error (see Eq.'s (6.2) and (6.3)):

$$p_{dg} = p_{sd} + p_{tt}.$$

Now we compute the second term in Eq. (6.4):

$$\begin{aligned}
&\sum_{m=1}^{\infty} Z^m \cdot P(M(t)=m) \\
&= \sum_{m=1}^{\infty} Z^m \sum_{k=m}^{\infty} P(M(t)=m \mid k)\cdot P(K(t)=k) \\
&= \sum_{m=1}^{\infty} Z^m \sum_{k=m}^{\infty} \binom{k}{m} (1-p_{dg}-p_{cf})^m \cdot {p_{dg}}^{k-m} \cdot P(K(t)=k). \qquad (6.7)
\end{aligned}$$

Using the results of Eq.'s (6.5), (6.6) and (6.7), Eq. (6.4) becomes:

$$\begin{aligned}
E[Z^M] &= \sum_{k=1}^{\infty} P(K(t)=k)\cdot(1-(1-p_{cf})^k) + \\
&\quad \sum_{m=0}^{\infty} Z^m \sum_{k=m}^{\infty} \binom{k}{m}(1-p_{dg}-p_{cf})^m \cdot {p_{dg}}^{k-m}\cdot P(K(t)=k) \\
&= \sum_{k=1}^{\infty} P(K(t)=k)\cdot(1-(1-p_{cf})^k) + \\
&\quad \sum_{k=0}^{\infty} P(K(t)=k) \sum_{m=0}^{k} Z^m \binom{k}{m}(1-p_{dg}-p_{cf})^m \cdot {p_{dg}}^{k-m} \\
&= \sum_{k=0}^{\infty} P(K(t)=k)\cdot((1-(1-p_{cf})^k) + (p_{dg}+(1-p_{dg}-p_{cf})Z)^k) \\
&= E\left[(1-(1-p_{cf})^{K(t)}) + (p_{dg}+(1-p_{dg}-p_{cf})\cdot Z)^{K(t)}\right]. \qquad (6.8)
\end{aligned}$$

The moment generating function enables the computation of the mean reward, Let $U = p_{dg} + (1-p_{dg}-p_{cf})\cdot Z$, then

$$\begin{aligned}
E[M] &= \left.\frac{d\,E[e^{sM}]}{d\,s}\right|_{s=0} \\
&= \left.\frac{d\,(E[1-(1-p_{cf})^{K(t)} + U^{K(t)}])}{d\,U}\cdot\frac{d\,U}{d\,Z}\cdot\frac{d\,Z}{d\,s}\right|_{s=0} \\
&= \frac{1-p_{dg}-p_{cf}}{1-p_{cf}}\cdot\sum_{k=0}^{\infty} k\cdot(1-p_{cf})^k\cdot P(K(t)=k) \\
&= \frac{1-p_{dg}-p_{cf}}{1-p_{cf}}\cdot E[K(t)\cdot(1-p_{cf})^{K(t)}] \qquad (6.9)
\end{aligned}$$

According to the Central Limit Theorem for Renewal Process [52], $K(t)$ is approximately normally distributed with the mean t/μ and the variance $t\sigma^2/\mu^3$

for large t (in which μ and σ^2 are the mean and variance of the renewal cycle time supplied by the performance sub-model), that is

$$\lim_{t\to\infty} P\left\{\frac{K(t)-t/\mu}{\sqrt{t\sigma^2/\mu^3}} < x\right\} = \Phi(x). \tag{6.10}$$

Let $\hat{\mu} = t/\mu$, $\hat{\sigma} = \sqrt{t\sigma^2/\mu^3}$, and $\alpha = \log(1-p_{cf})$, we have

$$\begin{aligned}
&E[K(t)\cdot(1-p_{cf})^{K(t)}] \\
&= E[(\hat{\sigma}X+\hat{\mu})e^{\alpha(\hat{\sigma}X+\hat{\mu})}] \\
&= e^{\alpha\hat{\mu}}(\hat{\sigma}E[Xe^{\alpha\hat{\sigma}X}] + \hat{\mu}E[e^{\alpha\hat{\sigma}X}]) \\
&= e^{\alpha\hat{\mu}}(\hat{\sigma}\int_{-\frac{\hat{\mu}}{\hat{\sigma}}}^{\infty} xe^{\alpha\hat{\sigma}x}\varphi(x)dx + \hat{\mu}\int_{-\frac{\hat{\mu}}{\hat{\sigma}}}^{\infty} e^{\alpha\hat{\sigma}x}\varphi(x)dx)
\end{aligned} \tag{6.11}$$

Applying the result to Eq. (6.9), we attain the expected reward.

6.6 NVP Model

Figure 6.6 shows the operation of an NVP scheme. The system has three functionally equivalent but independently developed programs (versions) and a decision function (voter). The supervisory system has a "watch dog timer" function, which functions similarly to the one in the Recovery Blocks scheme.

The system operates essentially as follows. The three versions start to execute at the same time. When all the three complete their execution, the decision function votes on their results. If there exists a majority representing a correct computation, the result from the majority will be output (the case corresponding to path 1 in Figure 6.6). If majority does not exist, the results will be suppressed (the case corresponding to path 3). The next iteration starts after either of the cases stated above. If there exists a majority representing similar erroneous com-

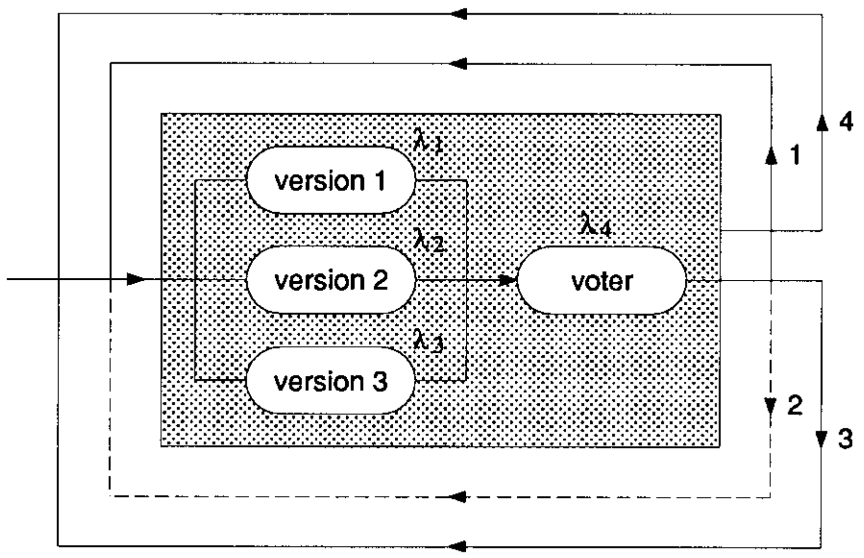


Figure 6.6: N-Version Programming Operation

putation, the erroneous result will be output (the case corresponding to path 2). From a stochastic process point of view, the iteration always resumes independent of outcome type; from the user point of view, however, a catastrophic failure (caused by an undetected error) leads to loss of service. Therefore, path 2 is represented by the dashed line in Figure 6.6. Finally, path 4 corresponds to the case that the execution of an iteration exceeds its real-time deadline such that the iteration is aborted and the next cycle starts. Since this scenario may occur at any stage of an iteration, path 4 starts from the shaded rectangular instead of any component encapsulated inside.

6.6.1 Dependability Sub-Model

We adapt the fault manifestation model for NVP in [41] to the lower layer as the dependability submodel. The fault classification and notations for probabilities of fault manifestation are illustrated in Table 6.3. Because the implementation of the voter is normally application independent, we omit the category "related fault in versions and voter."

Table 6.3: Fault Types and Notations for NVP

Fault Types	Probability of Manifestation
Related fault in the 3 versions	q_{3v}
Related fault in the 2 versions	q_{2v}
Independent fault in a version	q_{iv}
Independent fault in the voter	q_d

The derivation of the fault manifestation model is based on the following assumptions:

1. No compensation may take place between the errors of the versions and of the voter.
2. The probability that a majority exists among the version results, but the voter delivers a non-majority result, or no majority exists, but the voter delivers a good result from versions (error compensation), is negligible.

According to assumption 2, we decompose q_d into two parts, namely, q_{d1} and q_{d2}. The former is the probability that a voter fails to recognize an existing majority and suppresses the results (this corresponds to the case in which the decision algorithm in the voter is too "strict"); the latter is the probability that a voter fails to recognize the discrepancies among the versions and delivers an erroneous result (this corresponds to the case in which the decision algorithm is too "loose"). The detailed model, a Markov transition state diagram based on the above assumptions, is shown in Figure 6.7 [2]. The model is simpler than the one used in [41] because of assumption 2 and the decomposition of q_d. In the state diagram,

$$p = 1 - 3q_{iv}(1 - q_{iv})^2 - 3{q_{iv}}^2(1 - q_{iv}) - 3q_{2v} - q_{3v}.$$

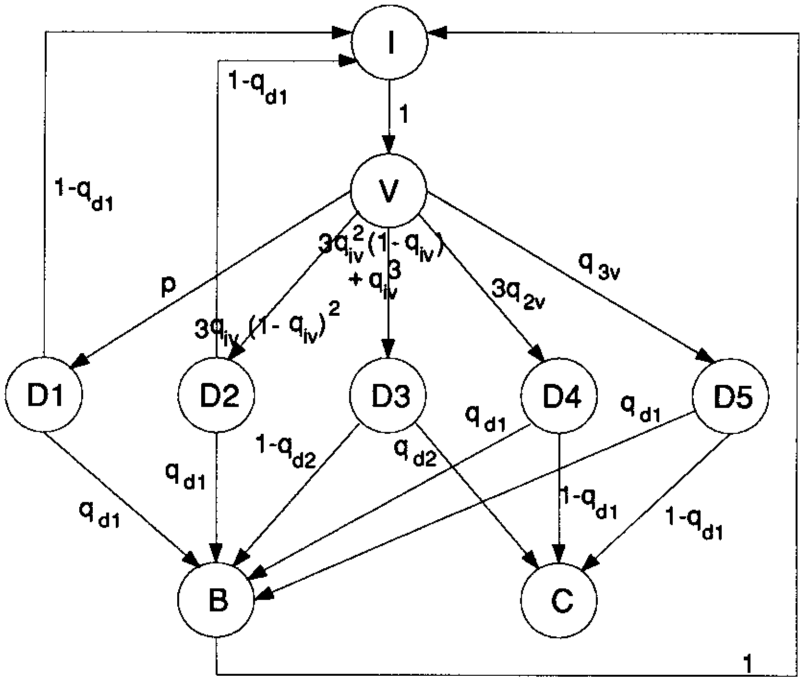


Figure 6.7: Dependability Sub-Model of NVP

The definitions of the states are shown in Table 6.4.

The partitioned states D_i account for the various types of faults manifested during version execution:

D_1: no fault manifestation; three acceptable results.

[2]In [41], the term $(1-q_{iv})$ is approximated to unity for simplicity. However, the motivation of fault tolerant software of this type is to achieve the dependability goal through design diversity such that the probability of independent errors need not be negligible (relative to correlated errors). Thus the term is presented in our formulation.

Table 6.4: State Definitions for NVP Dependability Model

States	Definition
I	initial state of an iteration
V	execution of versions
$\{D_i \mid i \in \{1,2,3,4,5\}\}$	execution of voter
B	benign failure
C	catastrophic failure

D_2: manifestation of an independent fault in one version; one independent erroneous result and two acceptable results.

D_3: manifestation of independent faults in two or three versions; three distinct results.

D_4: manifestation of correlated faults in two versions; two similar erroneous results.

D_5: manifestation of correlated faults in three versions; three similar erroneous results.

Subsequently, the fault-free behavior of the voter leads to the following transitions:

1. from D_1 and D_2 to I; that is, the voter evaluates two or three acceptable results.

2. from D_3 to B; that is, the voter evaluates three distinct results.

3. from D_4 and D_5 to C; that is, the voter evaluates two or three similar erroneous results.

The faulty behavior of the voter leads to the following transitions:

1. D_1, D_2, D_4 and D_5 to B; that is, the voter does not recognize an existing majority so suppresses the results.

2. D_3 to C; that is, the voter does not recognize the discrepancies and delivers an erroneous result.

As done for RB, let p_{cf} denote the probability of catastrophic failure (the case corresponding to state C in Figure 6.7 or to path 2 in Figure 6.6). And let p_{sd} denote the probability of benign failure (the case corresponding to state C in Figure 6.7 and to path 3 in Figure 6.6). Then from the Markov transition state diagram, we have:

$$p_{cf} = (3{q_{iv}}^2(1-q_{iv}) + {q_{iv}}^3) \cdot q_{d2} + 3q_{2v}(1-q_{d1}) + q_{3v}(1-q_{d1}). \quad (6.12)$$

We notice that the second term in the above equation is the dominant term. And

$$\begin{aligned} p_{sd} &= p \cdot q_{d1} + 3q_{iv}(1-q_{iv})^2 \cdot q_{d1} + (3{q_{iv}}^2(1-q_{iv}) + {q_{iv}}^3)(1-q_{d2}) + \\ &\quad 3q_{2v}q_{d1} + q_{3v}q_{d1} \\ &= (1 - 3{q_{iv}}^2(1-q_{iv})) \cdot q_{d1} + (3{q_{iv}}^2(1-q_{iv}) + {q_{iv}}^3)(1-q_{d2}). \quad (6.13) \end{aligned}$$

We notice that the probability of benign failure is comprised by the probability of voter error type one (ignorance of a majority) and the probability of independent errors in a version (a second order factor). The other type of benign failure is the detected real-time deadline violation, which corresponds to path 4 in Figure 6.6. The analysis of benign failure of this type is represented in the next section.

6.6.2 Performance Sub-Model

We assume that the execution time of the three versions in the NVP scheme are independently and exponentially distributed with parameters λ_1, λ_2 and λ_3. Figure 6.8 depicts the renewal process $K(t)$, which is derived from Figure 6.6. By definition, the random variable $K(t)$ is independent of outcome type, hence paths 1, 2 and 3 in Figure 6.6 are aggregated into a single path in Figure 6.8.

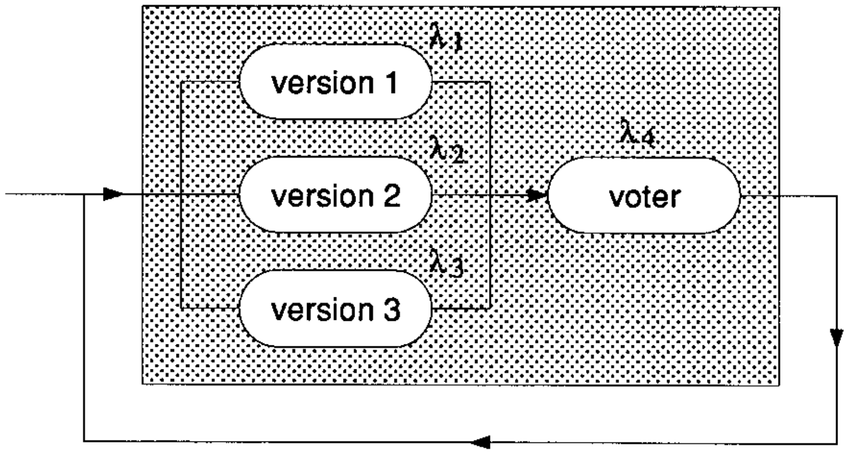


Figure 6.8: Performance Sub-Model of NVP

Let the time for parallel version execution be denoted as Y_v. And let the execution time of the first version, the second version and the third version be denoted as Y_1, Y_2 and Y_3, respectively. Then,

$$Y_v = MAX[Y_1, Y_2, Y_3].$$

And the PDF of Y_v is:

$$\begin{aligned} G(y) &= P(Y_v \leq y) \\ &= P(Y_1 \leq y,\, Y_2 \leq y,\, Y_3 \leq y) \\ &= P(Y_1 \leq y) \cdot P(Y_2 \leq y) \cdot P(Y_3 \leq y) \end{aligned}$$

Thus we can obtain the pdf of Y_v, that is, $g(y)$. Let the Laplace transform of $g(y)$ be denoted as $G^*(S)$ and the transform of the pdf of the voter execution time (also distributed exponentially) be denoted as $H^*(s)$. Since the parallel version execution is followed by the voter execution, the transform of the cycle time of the renewal process is

$$F^*(S) = G^*(S) \cdot H^*(S).$$

Next the density function of the renewal cycle time Y can be obtained through inverse transform:

$$F^*(S) \Leftrightarrow f(y).$$

Due to a deadline τ (the real-time constraint), the iteration time Y follows the distribution defined by $f(y)$ if the iteration completes before the deadline; otherwise it stops at τ, which corresponds to path 4 in Figure 6.6 and can be represented by the unit impulse at τ scaled by the probability of deadline violation. The actual density function of Y, and the corresponding mean and variance are attained in the same manner as we did for RB (Section 6.5.2).

6.6.3 Performability Model

Figure 6.9 depicts the performability model at the upper layer in a similar manner as for the RB scheme.

Contrasting Figure 6.6 to Figure 6.9, path 1 in Figure 6.6 corresponds to the path marked $(1 - p_{dg} - p_{cf})$ in Figure 6.9, path 2 corresponds to the path marked p_{cf}, and paths 3 and 4 collapse to the path marked p_{dg}. Thus, after integrating the information supplied by the lower layer models, we have arrived at an upper layer model identical to that for RB scheme. It follows that the moment generating function derived from the performability model of NVP is the same as that of

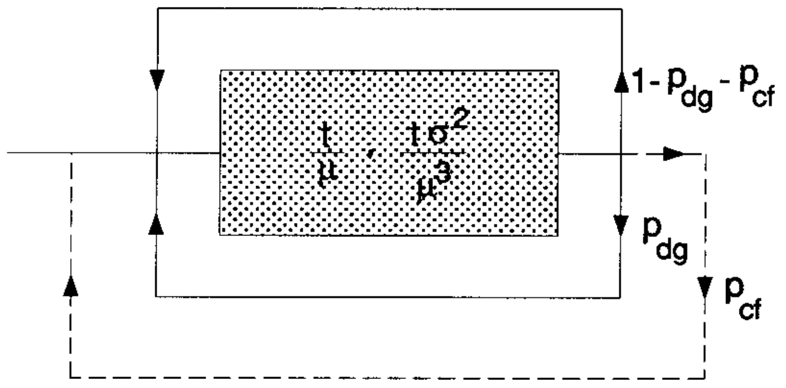


Figure 6.9: Performability Model of NVP

RB as illustrated in Section 6.5.3.

6.7 Results Analysis

6.7.1 Comparisons between RB and NVP

We computed the expected reward of a 10 hour mission for both RB and NVP. In order to make coherent comparisons, we assume that:

1. Except for the application independent component, the voter in NVP, the probabilities of independent errors in the individual components, that is, P, S and AT for RB and each version for NVP, are essentially equal.

2. The probabilities of similar errors between (two or three) components for RB and NVP are essentially equal.

The extent to which AT detects errors in an alternative is often referred as "coverage." The execution time of AT is a direct function of coverage; whereas the probability of faults in AT is a direct function of its execution time [36].

For simplicity and consistency, we assume full coverage for AT in this evaluation. Accordingly, the probability of independent errors in AT is assumed to be comparable with that of an alternative or a version; the probabilities of correlated faults between P and AT, and between S and AT are assumed to be comparable with those between two alternatives or two versions; the execution time distribution of AT is assumed to be comparable with that of an alternative or a version.

Table 6.5 defines the notations for system parameters used in our evaluation. The assignments of system parameters for RB and NVP are shown in Table 6.6 and 6.7 respectively. The detailed computation, including Laplace transforms and derivations of density functions, are implemented in Mathematica. [3] The sample Mathematica programs and output files are shown in Appendix D.

Figure 6.10 shows the comparison of the expected reward between RB and NVP for a 10 hour mission. The reward is the expected number of successful iterations for the time interval t (which equals to 3.6 10^7 milliseconds) according to our definition of the performance variable.

The upper and lower curves in Figure 6.10 are the expected reward as functions of the probability of correlated errors between any two components in RB and NVP, respectively. The results show that RB outperforms NVP. The formulation of the expected reward is (Eq. (6.9)):

$$E[M] = \frac{1 - p_{dg} - p_{cf}}{1 - p_{cf}} \cdot E[K(t) \cdot (1 - p_{cf})^{K(t)}],$$

which suggests that we should investigate the probabilistic failure behavior and the random variable $K(t)$. Accordingly, we analyze the deficiencies of NVP by tracing the system behavior represented by the sub-models in the lower layer.

[3]Mathematica is a registered trademark of the Wolfram Research Inc.

Table 6.5: Notations for Parameters

Parameter	Definition
λ_p	execution rate of the primary
λ_s	execution rate of the secondary
λ_t	execution rate of the acceptance test
λ_1	execution rate of the first version
λ_2	execution rate of the second version
λ_3	execution rate of the third version
λ_4	execution rate of the voter
q_{ps}	probability of related errors between P and S
q_{pt}	probability of related errors between P and AT
q_{st}	probability of related errors between S and AT
q_{pst}	probability of related errors between P, S and AT
q_{2v}	probability of related errors between two versions
q_{3v}	probability of related errors between three versions
q_p	probability of independent errors in primary
q_s	probability of independent errors in secondary
q_t	probability of independent errors in acceptance test
q_{iv}	probability of independent errors the i^{th} version
q_{d_1}	probability of type 1 independent errors in the voter [a]
q_{d_2}	probability of type 2 independent errors in the voter [b]
t	length of mission time
τ	deadline for an iteration

[a]see definition in Section 6.6.1
[b]see definition in Section 6.6.1

Table 6.6: Assignments of Parameters for RB

Parameter	Value
λ_p	$\frac{1}{5}$
λ_s	$\frac{1}{8}$
λ_t	$\frac{1}{5}$
q_{ps}	variable
q_{pt}	variable
q_{st}	variable
q_{pst}	10^{-10}
q_p	0.0001
q_s	0.0001
q_t	0.0001
t	$3.6\ 10^7$ (ms)
τ	30 (ms)

Table 6.7: Assignments of Parameters for NVP

Parameter	Value
λ_1	$\frac{1}{5}$
λ_2	$\frac{1}{6}$
λ_3	$\frac{1}{8}$
λ_4	$\frac{1}{0.5}$
q_{2v}	variable
q_{3v}	10^{-10}
q_{iv}	0.0001
q_{d_1}	10^{-9}
q_{d_2}	10^{-9}
t	$3.6\ 10^7$ (ms)
τ	30 (ms)

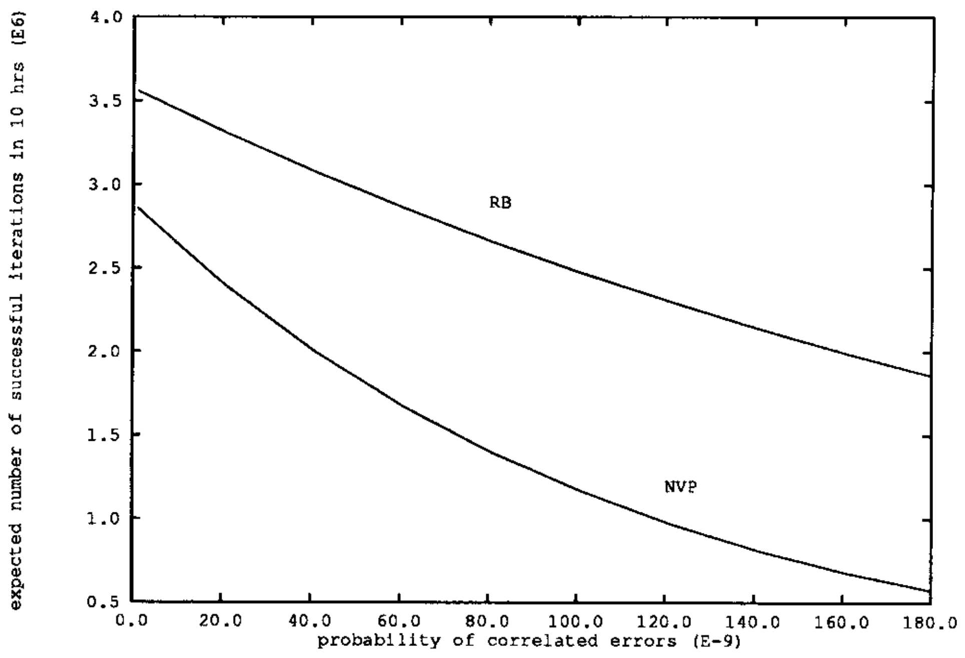


Figure 6.10: Comparisons of RB and NVP

From the dependability submodels (Section 6.5.1 and Section 6.6.1), we can see that for both RB and NVP, the probability of catastrophic failure (p_{cf}, see Eq.'s (6.1) and (6.12)) is dominated by the probability of correlated errors between the components. In the RB scheme, the probability of correlated errors between two components comprises the terms q_{pt}, q_{st} and q_{ps}. We notice that among them only q_{pt} completely contributes to p_{cf}, while q_{ps} is not responsible for catastrophic failure. As to q_{st}, it causes a catastrophic failure under the condition that AT rejects P. The probability that AT rejects P is a factor which reduces the contribution of q_{st} to p_{cf}. In the NVP scheme, on the other hand, the probability of correlated errors between any two components directly contributes to p_{cf}. Therefore, p_{cf} of NVP is approximately three times greater than that of RB.

By looking into the performance submodels (Section 6.5.2 and Section 6.6.2), we can see that: 1) for RB, the mean iteration time is dominated by the mean execution time of P and AT, while 2) for NVP, the mean iteration time is lengthened due to the fact that synchronization requires waiting for the slowest version — a severe performance penalty.

Form Figure 6.10 we notice the following: the difference between the RB and NVP curves at the low correlated error probability end (left hand side) is smaller than the difference at the high correlated error probability end (right hand side). The underlying factors for this scenario are:

- when the probability of correlated errors is low, the difference is primarily caused by the difference in the performance cost.
- when the probability of correlated errors is high, the difference is not only a result of the performance cost, but is also amplified by the fact that NVP failure behavior is more sensitive to the correlated errors between components.

6.7.2 Comparisons between Fault-Tolerant Software and Non-Fault-Tolerant Software

Figure 6.11 shows the expected reward in a 10 hour mission for non-fault-tolerant software. We assume that the program execution time is exponentially distributed with a mean $(\frac{1}{\lambda})$ of 5 milliseconds. Its error probability q is kept as a variable. Since conventional software does not apply error detection or recovery mechanisms, manifestation of program faults will always cause catastrophic failures. The computation is straight forward. We show the formulation of the expected reward below, without derivation.

$$E[M] = e^{-\lambda t q} \cdot \lambda t \cdot (1-q)$$

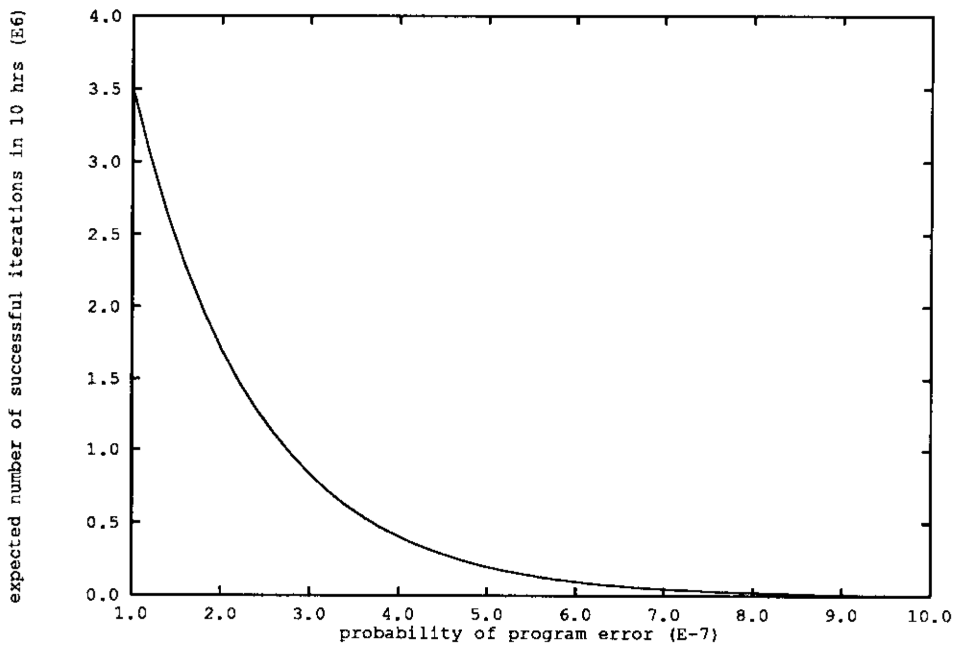


Figure 6.11: Expected Reward for Non-Fault-Tolerant Software

Notice that the expected reward approaches zero well before the program error probability reaches 10^{-4}. This is the value of the probability of independent errors in a component used for the fault tolerant software evaluations in Figure 6.10. In order to achieve a reward in the range compatible with that accomplished by RB with correlated error probability on the order of 10^{-9} to 10^{-7}, or that accomplished by NVP with correlated error probability on the order of 10^{-9} to 10^{-8}, the error probability of the non-fault-tolerant program needs to be as low as 10^{-7}. Therefore, under the condition of low correlated error probability, the software fault tolerance techniques significantly improve performability.

6.8 Alternative NVP Model

6.8.1 An Alternative Approach to NVP Operation

From the above analysis, we recognize that NVP in general is inferior on performability issues compared with RB, because of the severe performance penalty from synchronization and the fact that NVP is more vulnerable to catastrophic failures due to correlated errors. The performability analysis indicates possible directions to improve the NVP scheme. One way is to improve the performance by modifying the use of the computational redundancy in an operational context. In the alternative NVP scheme, the decision function starts voting (comparison) when any two of the versions complete their executions and wait for the slowest one only if the comparison indicates a discrepancy. If the two versions agree, the consensus result is output, and the third version is aborted. Subsequently,

- for an open loop application, the next iteration starts with new input data;
- for a closed loop application, the internal state of the aborted version is re-initialized using the internal state of a successful version, then the next iteration starts with new input data.

If the two versions disagree, the system will wait for the third version to complete. Upon the third version's completion, the second vote takes place on the results submitted by all the three versions. Our motivation is to make the performance of NVP be dominated by the faster versions instead of the slowest version, while keeping the slowest version as a "back-up" for the dependability objective. We assume that the first and the second voters would be implemented with different decision algorithms in order to minimize the probability that a voter outvotes acceptable results or passes erroneous results. Further, we assume that the internal state of a program in a closed loop application is small, such that its restoration takes negligible time.

Figure 6.12 shows the operation of the alternative NVP scheme which consists of three versions and two voters.

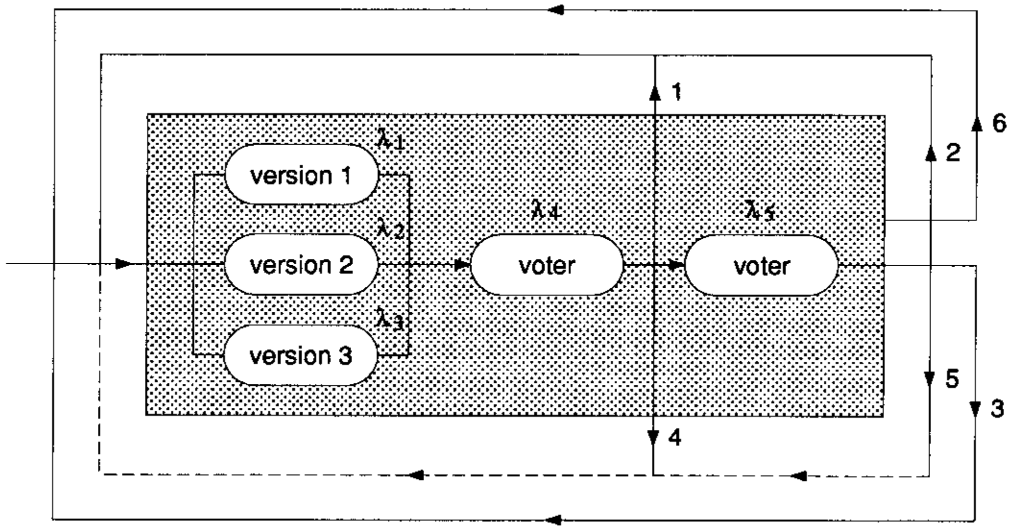


Figure 6.12: Alternative NVP Operation

The system operates essentially as follows. The three versions start to execute at the same time. When any two of the versions complete, the first voter compares their results. If both of them compute correctly, the consensus result will be output according to the decision algorithm (the case corresponding to path 1 in Figure 6.12). If the two versions produce similar erroneous result, the voter outputs the consensus erroneous result (the case corresponding to path 4 in Figure 6.12). If discrepancy in the two versions' results is detected by the voter, the system waits for the result of the third version. The second voter takes action when the third result is submitted. If there exists a majority representing correct computation, the result from the majority will be output (the case corresponding to path 2 in Figure 6.12). If a majority does not exist, the results will be suppressed (the case corresponding to path 3). If there exists a majority representing similar erroneous computation, the erroneous result will be output (the case corresponding to path 5). From a stochastic process point of view, the iteration always resumes independent of outcome type; from a user point of view, however, a catastrophic failure (caused by an undetected error) leads to loss of service. Therefore, path 5 is represented by the dashed line in Figure 6.12. Finally, path 6 corresponds to the case that the execution of an iteration exceeds its real-time deadline, such that the iteration is aborted and the next cycle starts. Since this scenario may occur at any stage of an iteration, path 6 starts from the shaded rectangular instead of any component encapsulated inside.

We have assessed the performability improvement from this alternative approach, which is described in the following sections.

6.8.2 Dependability Sub-Model

The fault classification and notations for probabilities of fault manifestation are illustrated in Table 6.8. We assume that the probabilities of fault manifestation for the two voters are equivalent. That is,

$$q_d^1 = q_d^2 = q_d.$$

As explained in Section 6.6.1, we decompose q_d into two parts, namely, q_{d1} and q_{d2}. The former is the probability that a voter fails to recognize an existing majority (or consensus between two versions) and suppresses the results; the latter is the probability that a voter fails to recognize the discrepancies among the versions and delivers an erroneous result.

Table 6.8: Fault Types and Notations for new-NVP

Fault Types	Probability of Manifestation
Related fault in the 3 versions	q_{3v}
Related fault in the 2 versions	q_{2v}
Independent fault in a version	q_{iv}
Independent fault in the voter 1	q_d^1
Independent fault in the voter 2	q_d^2

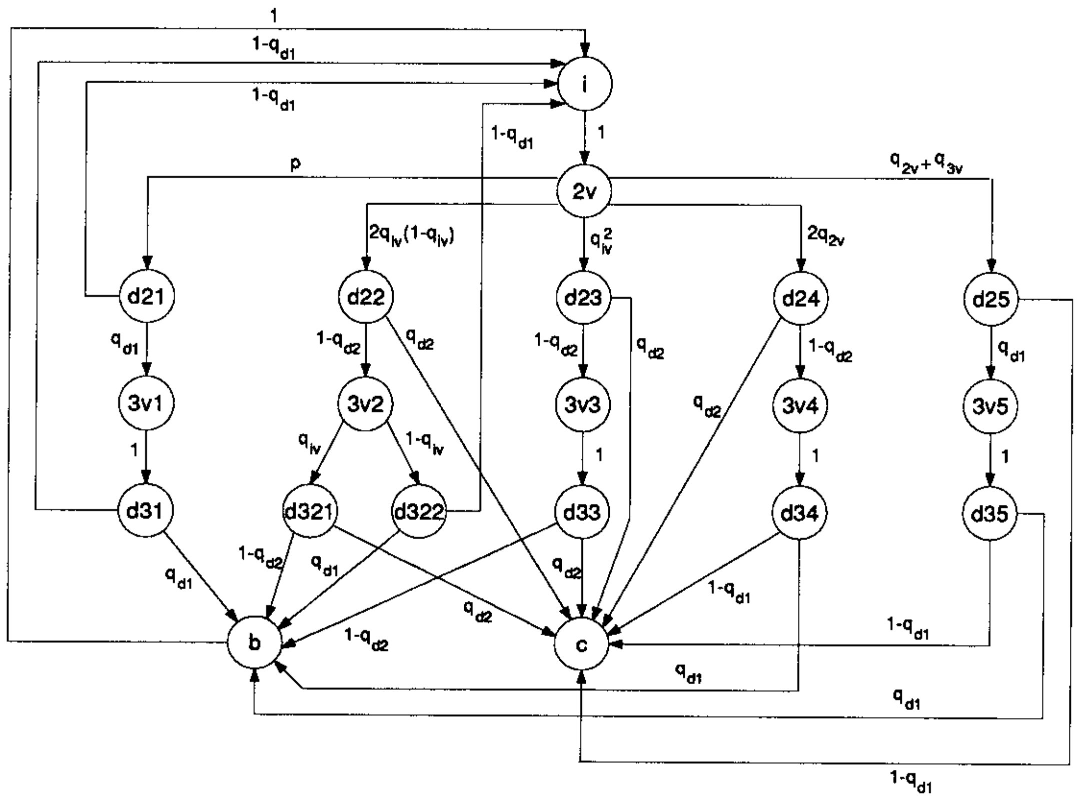


Figure 6.13: Dependability Sub-Model of Alternative NVP

We apply the assumptions for NVP stated in Section 6.6.1 to the alternative NVP. The Markov transition state diagram is shown in Figure 6.13, where

$$p = 1 - 2q_{iv}(1 - q_{iv}) - q_{iv}^2 - 3q_{2v} - q_{3v}.$$

The definitions of the states are shown in Table 6.9.

Table 6.9: State Definitions for new-NVP Dependability Model

States	Definition
I	initial state of an iteration
$2V$	first stage of version execution (waiting for the first two versions' completion)
$\{D_{2i} \mid i \in \{1,2,3,4,5\}\}$	execution of voter after two versions complete
$\{3V_i \mid i \in \{1,2,3,4,5\}\}$	second stage of version execution (waiting for the last version's completion)
$\{D_{3i} \mid i \in \{1,2,3,4,5\}\}$	execution of voter after three versions complete
B	benign failure
C	catastrophic failure

The partitioned states D_{2i} correspond to the various types of faults manifested during the first stage of version execution:

D_{21}: no fault manifestation; two acceptable results.

D_{22}: manifestation of an independent fault in one version; one independent erroneous result and one acceptable result.

D_{23}: manifestation of independent faults in the two versions; two distinct results.

D_{24}: potential manifestation of correlated faults in two versions — one is completed and the other is still in execution; one erroneous result and one acceptable or independent erroneous result.

D_{25}: manifestation of correlated faults in two or three versions; two similar erroneous results.

Subsequently, the fault-free behavior of the voter leads to the following transitions:

1. from D_{21} to I; that is, the voter evaluates two acceptable results.
2. from D_{22}, D_{23} and D_{24} to $3V_2$, $3V_3$ and $3V_4$, respectively; that is, the voter evaluates two distinct results.
3. from D_{25} to C; that is, the voter evaluates two similar erroneous results.

The faulty behavior of the voter leads to the following transitions:

1. D_{21} to $3V_1$; that is, the voter does not recognize the consensus between the two versions, so decides to wait for the third version.
2. D_{22}, D_{23}, and D_{24} to C; that is, the voter does not recognize the discrepancies and delivers an erroneous result.
3. D_{25} to $3V_5$; that is, the voter does not recognize the consensus between the erroneous results and decides to wait for the third version.

The partitioned states D_{3i} correspond to the various outcomes of the second stage of version execution.

D_{31}: at least two acceptable results.

D_{32}: D_{321} corresponds to the manifestation of an independent fault in the third version; two independent erroneous results and one acceptable result. D_{322} corresponds to no fault manifestation in the third version; one independent erroneous result and two acceptable results.

D_{33}: three distinct results.

D_{34}: manifestation of correlated faults in two versions — one completed at the first stage and the other completed at the second stage; two similar erroneous results and one acceptable or independent erroneous result.

D_{35}: two or three similar erroneous results.

Subsequently, the fault-free behavior of the second voter leads to the following transitions:

1. from D_{31} and D_{322} to I; that is, the voter evaluates two or three acceptable results.
2. from D_{321} and D_{33} to B; that is, the voter evaluates three distinct results.
3. from D_{34} and D_{35} to C; that is, the voter evaluates two or three similar erroneous results.

The faulty behavior of the voter leads to the following transitions:

1. D_{31}, D_{322}, D_{34} and D_{35} to B; that is, the voter does not recognize the consensus between the two or three versions and suppresses the results.
2. D_{321} and D_{33} to C; that is, the voter does not recognize the discrepancies and delivers an erroneous result.

As done for RB and NVP, let p_{cf} denote the probability of catastrophic failure (the case corresponding to state C in Figure 6.13 and to paths 4 and 5 in Figure 6.12). Let p_{sd} denote the probability of benign failure (the case corresponding to state B in Figure 6.13 and to path 3 in Figure 6.12). From the Markov transition state diagram, we have:

$$\begin{aligned}
p_{cf} &= 2q_{iv}(1-q_{iv})q_{d2} + 2q_{iv}^2(1-q_{iv})(1-q_{d2})\cdot q_{d2} + {q_{iv}}^2 \cdot q_{d2} + \\
&\quad {q_{iv}}^2(1-q_{d2})q_{d2} + 2q_{2v}\cdot q_{d2} + 2q_{2v}(1-q_{d2})(1-q_{d1}) + \\
&\quad (q_{2v}+q_{3v})(1-q_{d1}) + (q_{2v}+q_{3v})q_{d1}(1-q_{d1}) \\
&= 2q_{iv}(1-q_{iv})q_{d2}(1+q_{iv}-q_{d2}q_{iv}) + {q_{iv}}^2 q_{d2}(2-q_{d2}) + \\
&\quad q_{2v}(1-q_{d1}-q_{d1}\cdot q_{d2}) + q_{2v}(1-q_{d1}^2) + q_{3v}(1-q_{d1}^2) \\
&\approx 2q_{iv}\cdot q_{d2}(1-{q_{iv}}^2) + {q_{iv}}^2\cdot q_{d2}(2-q_{d2}) + \\
&\quad q_{2v}(1-q_{d1})(3+q_{d1}) + q_{3v}. \qquad (6.14)
\end{aligned}$$

It is clear that the probabilities of correlated faults dominate the likelihood of catastrophic failure. And we have:

$$\begin{aligned}
p_{sd} &= p\cdot {q_{d1}}^2 + 2{q_{iv}}^2(1-q_{iv})(1-q_{d2})^2 + 2q_{iv}(1-q_{iv})^2(1-q_{d2})q_{d1} + \\
&\quad {q_{iv}}^2(1-q_{d2})^2 + 2q_{2v}(1-q_{d2})q_{d1} + (q_{2v}+q_{3V})q_{d1}^2 \\
&= p\cdot q_{d1}^2 - 2q_{iv}^3(1-q_{d2})(1-q_{d1}-q_{d2}) + q_{iv}^2(1-q_{d2})(3(1-q_{d2})-4q_{d1}) + \\
&\quad 2q_{iv}(1-q_{d2})q_{d1} + 2q_{2v}(1-q_{d2})q_{d1} + (q_{2v}+q_{3v})q_{d1}^2 \\
&\approx 2q_{iv}(1-q_{d2})q_{d1} + q_{iv}^2(1-q_{d2})(3(1-q_{d2})-4q_{d1}) - \\
&\quad 2q_{iv}^3(1-q_{d2})(1-q_{d1}-q_{d2}). \qquad (6.15)
\end{aligned}$$

The above equation reveals that the probability of benign failure will be low if the probability of independent faults in a version is reasonably low. The other type of benign failure is the detected real-time deadline violation, which corresponds to path 6 in Figure 6.12. The analysis of benign failure of this type is presented in the next section.

6.8.3 Performance Sub-Model

We assume that the execution time of the first, the second and the third version are independently and exponentially distributed, with the parameters λ_1, λ_2 and λ_3, respectively. Assume also that the execution time of the first and second voter are independently and exponentially distributed, with parameters λ_4 and λ_5, respectively. Figure 6.14 depicts the renewal process $K(t)$, which is derived from Figure 6.12. Since the definition of the random variable $K(t)$ is independent of outcome type, paths 1 and 4 in Figure 6.12 are aggregated into a single path marked p_2 in Figure 6.14; likewise, paths 2, 3 and 5 in Figure 6.12 are aggregated into a single path marked $(1 - p_2)$. The notation p_2 represents the probability that an iteration completes upon voting on two consensus-correct or consensus-erroneous results. From the dependability sub-model (Figure 6.13), we have,

$$\begin{aligned} p_2 &= p(1 - q_{d1}) + 2q_{iv}(1 - q_{iv})q_{d2} + q_{iv}^2 \cdot q_{d2} + 2q_{2v} \cdot q_{d2} + (q_{2v} + q_{3v})(1 - q_{d1}) \\ &= (p + q_{2v} + q_{3v})(1 - q_{d1}) + (2q_{iv} - q_{iv}^2 + 2q_{2v})q_{d2}. \end{aligned} \tag{6.16}$$

For the sake of illustration, we introduce the following notations. Let Y denote the renewal cycle time. Let Y_1, Y_2 and Y_3 denote the time for any one of the versions, any two of the versions, and all three of the versions to complete, respectively. And let Y_{d1} and Y_{d2} denote the time for the first and second voter to complete, respectively. Let Y^I denote the sum of the time for any two versions to complete and the time for the first voter to complete. That is,

$$Y^I = Y_2 + Y_{d1}.$$

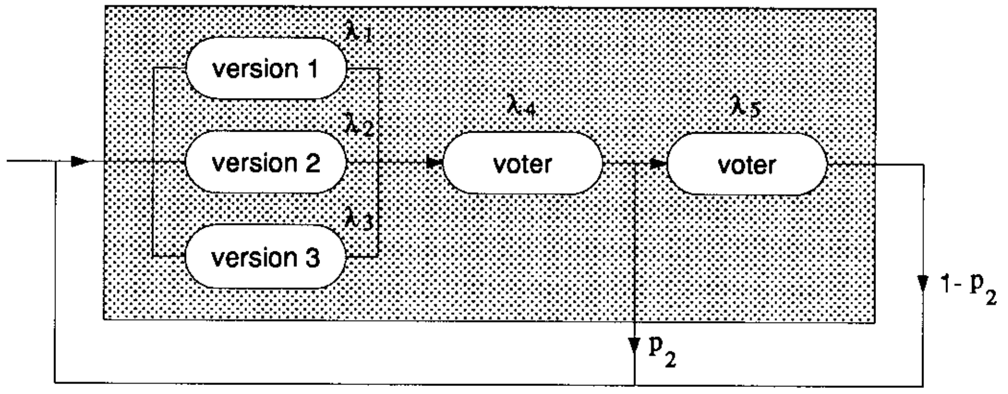


Figure 6.14: Performance Sub-Model of Alternative NVP

Let Y^{II} denote the maximum of Y^I and Y_3 (the time when all three versions complete their executions).[4] That is,

$$Y^{II} = MAX[Y^I, Y_3].$$

Let Y^{III} denote the sum of Y^{II} and Y_{d2}. That is,

$$Y^{III} = Y^{II} + Y_{d2}.$$

Let the pdf of Y be denoted as $f(y)$. According to the alternative NVP scheme,

$$f(y) = p_2 \cdot f_1(y) + p_2 \cdot f_2(y), \tag{6.17}$$

where p_2 is the probability that an iteration completes at the time when the first voter is done (Eq. (6.16)), $f_1(y)$ is the pdf of Y^I, and $f_2(y)$ is the pdf of Y^{III}. The derivation of $f(y)$ is as follows.

The PDF of Y_2 is:

$$G_1(y) = P(Y_2 < y) = P(\text{at least 2 programs completed in } (0, y)).$$

[4]This notation is necessary since the slowest version may complete during or after the first voter's execution.

That is

$$G_1(y) = P(Y_1 < y)P(Y_2 < y)P(Y_3 > y) + P(Y_1 < y)P(Y_2 < y)P(Y_3 < y). \quad (6.18)$$

Since the execution time of the three programs are independently and exponentially distributed with the parameters λ_1, λ_2 and λ_3, the equation (6.18) becomes

$$\begin{aligned} G_1(y) &= (1-e^{-\lambda_1 y})(1-e^{-\lambda_2 y})e^{-\lambda_3 y} + (1-e^{-\lambda_2 y})(1-e^{-\lambda_3 y})e^{-\lambda_1 y} + \\ &\quad (1-e^{-\lambda_1 y})(1-e^{-\lambda_3 y})e^{-\lambda_2 y} + (1-e^{-\lambda_1 y})(1-e^{-\lambda_2 y})(1-e^{-\lambda_3 y}). \end{aligned}$$

The pdf of Y_2 is the derivative of $G_1(y)$, denoted as $g_1(y)$. The pdf of the execution time for the first voter is:

$$h_1(y) = \lambda_4 e^{-\lambda_4 y}.$$

Let the Laplace transforms of $g_1(y)$ and $h_1(y)$ be denoted as $G_1^*(S)$ and $H_1^*(S)$, respectively. Then the Laplace transform of $f_1(y)$ is

$$F_1^*(S) = G_1^*(S) \cdot H_1^*(S).$$

Since $f_1(y)$, the pdf of Y^I, is attainable from the inverse Laplace transform of $F_1^*(S)$, we can derive the PDF of Y^I, i.e., $F_1(y)$.

Let $V(y)$ denote the PDF of Y_3, that is,

$$V(y) = (1-e^{-\lambda_1 y})(1-e^{-\lambda_2 y})(1-e^{-\lambda_3 y}).$$

Let $G_2(y)$ denote the PDF of Y^{II}, that is,

$$G_2(y) = F_1(y) \cdot V(y).$$

The pdf of Y^{II}, $g_2(y)$, is then attainable. Let $h_2(y)$ denote the pdf of the execution time of the second voter, that is,

$$h_2(y) = \lambda_5 e^{-\lambda_5 y},$$

and let the Laplace transform of $g_2(y)$ and $h_2(y)$ be denoted as $G_2^*(S)$ and $H_2^*(S)$, respectively, then we have,

$$F_2^*(S) = G_2^*(S) \cdot H_2^*(S).$$

which is the Laplace transform of Y^{III}. Then we can obtain the Laplace transform of Y, the renewal cycle time:

$$F^*(S) = p_2 \cdot F_1^*(S) + (1 - p_2) \cdot F_2^*(S).$$

where p_2 is derived from Eq. (6.16). And the density function of the renewal cycle time can be obtained through the inverse transform:

$$F^*(S) \Leftrightarrow f(y).$$

Due to a deadline τ, the iteration time Y follows the distribution defined by $f(y)$ if the iteration completes before the deadline; otherwise it stops at τ, which corresponds to path 6 in Figure 6.12 and can be represented by the unit impulse at τ scaled by the probability of deadline violation. The actual density function of iteration time Y (with the consideration of real-time constraint), and the corresponding mean and variance are solved in the same manner as we did for RB (Section 6.5.2).

6.8.4 Performability Model

Figure 6.15 depicts the performability model at the upper layer in a similar manner as for the RB and NVP schemes.

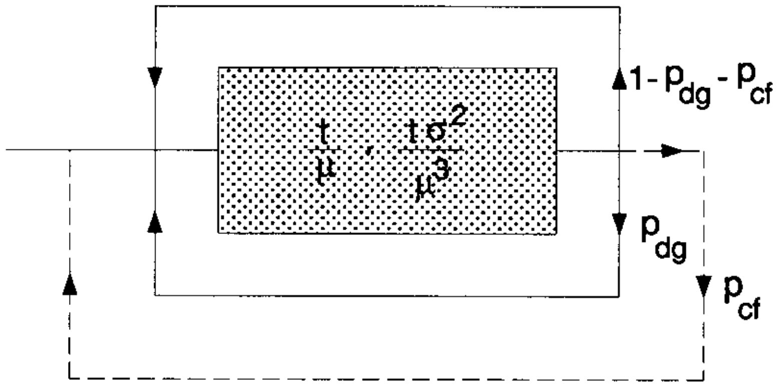


Figure 6.15: Performability Model of Alternative NVP

Contrasting Figure 6.12 to Figure 6.15, paths 1 and 2 in Figure 6.12 collapse to the path marked $(1 - p_{dg} - p_{cf})$ in Figure 6.15, paths 4 and 5 collapse to the path marked p_{cf}, and paths 3 and 6 collapse to the path marked p_{dg}.

Thus, integrating the information supplied by the lower layer models leads to an upper layer model identical to that for RB and NVP. It follows that the moment generating function derived from the performability model of the alternative NVP is the same as that for RB and NVP, as illustrated in Section 6.5.3.

6.9 Discussion

For sake of illustration, we use the abbreviation *new-NVP* for "the alternative approach to NVP" in the rest of the chapter.

Our evaluation results show that the performance penalty due to version synchronization is greatly reduced in the new-NVP scheme. This improvement is contributed to by the fact that the version which takes the longest time to complete becomes a back-up. The performance of an iteration is independent of this dynamic back-up version, unless discrepancy is detected in the two versions

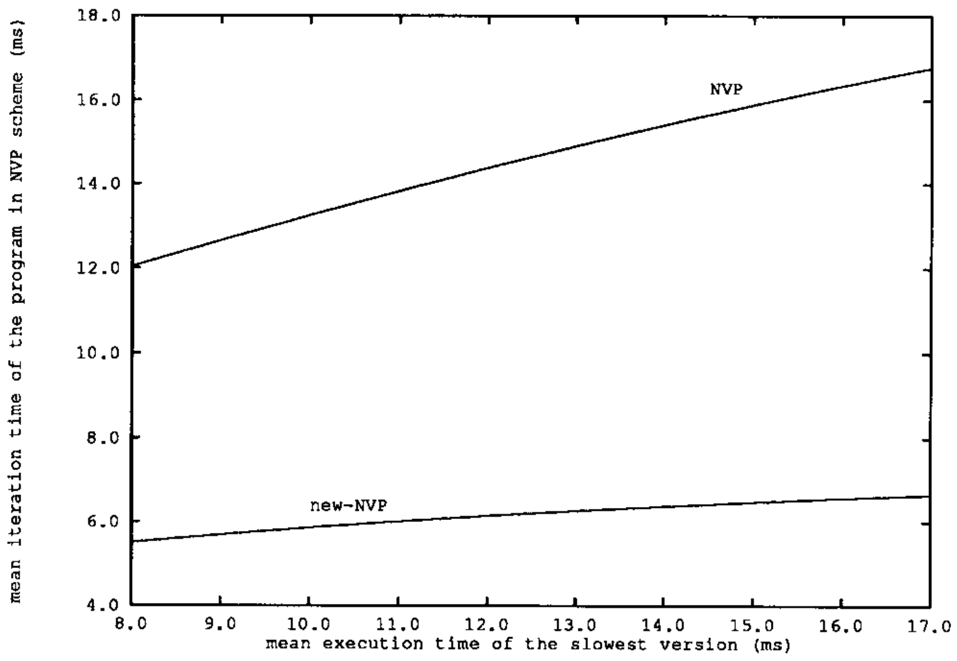


Figure 6.16: Sensitivity to the Mean Execution Time of the Slowest Version

completed first. In other words, in the new-NVP scheme, the iteration time is dominated by the two faster versions instead of the slowest one. As shown in Figure 6.16, the mean iteration time of the new-NVP scheme is insensitive to the mean execution time of the slowest version (here the "slowest" is identified by the mean and is used in a static sense), compared with NVP.

We have evaluated the expected reward for a 10 hour mission with the new-NVP. For coherent comparisons, we use the same assumptions and the same parameter assignments as we used for NVP.

Figure 6.17 shows the expected reward for RB, NVP and new-NVP as a function of the probability of correlated errors.

We have the following observations:

1. In the high dependability region (which corresponds to the low probability

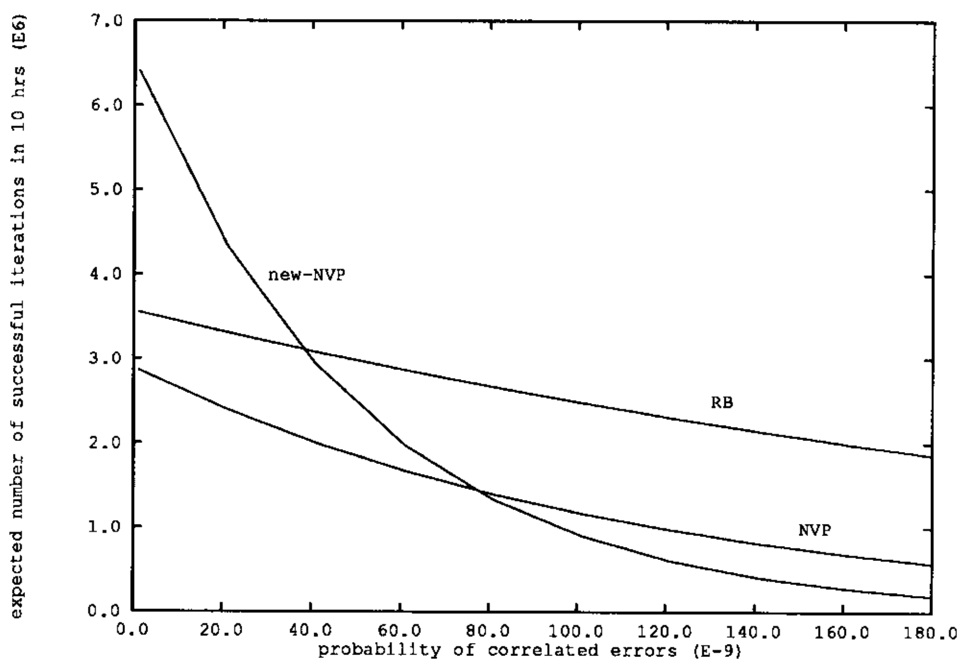


Figure 6.17: Comparisons of RB, NVP and Alternative NVP

of correlated errors), ranging from the beginning of the X-axis to the cross point of the new-NVP and RB curves, new-NVP outperforms both RB and NVP. This is because performance is the primary factor influencing the reward in this region, so that new-NVP becomes superior due to its outstanding performance.

2. In the moderate dependability region (which corresponds to the moderate probability of correlated errors), ranging from the cross point of the new-NVP and RB curves to the cross point of the new-NVP and NVP, the reward of new-NVP drops below RB but still above NVP. This is because the dependability influence becomes visible in this region so that the performance merit of new-NVP can no longer compensate for its dependability deficiency.

3. In the low dependability region (which corresponds to the high probability of correlated errors), ranging from the cross point of the new-NVP and NVP curves to the end of the X-axis, new-NVP becomes inferior to both RB and NVP. Its reward is even poorer than NVP for the following reason: the new-NVP executes at a higher rate while it does not incorporate any major dependability improvement, so it has a greater chance of running into an input which activates the correlated faults between the versions and leads to loss of the mission.

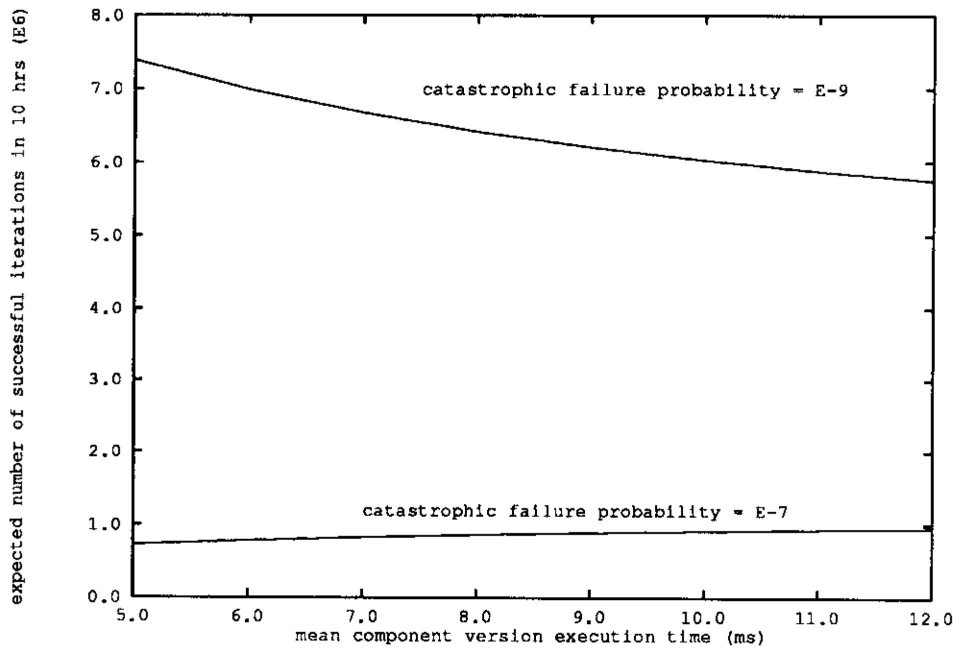


Figure 6.18: Impact of Component Performance

Figure 6.18 illustrates the impact of component performance on expected reward for the new-NVP. The expected reward is a function of the mean execution time of a component version in the new-NVP scheme. The upper curve is evaluated under the condition of a low catastrophic failure probability, while the lower curve is evaluated under the condition of a high catastrophic failure probability.

We can see that the expected reward increases when the component version performance improves for the upper curve; on the other hand, the expected reward decreases when component version performance improves for the lower curve. This phenomenon can be explained as follows. When the dependability is not sufficiently high, a higher iteration rate in general results in a poorer expected reward because of the higher probability that an undetected error will trigger the alteration of the performance level from "proper" to "improper". Its implication in software engineering practice is: perfective maintenance is beneficial only when dependability reaches a certain level, otherwise corrective maintenance should be given priority.

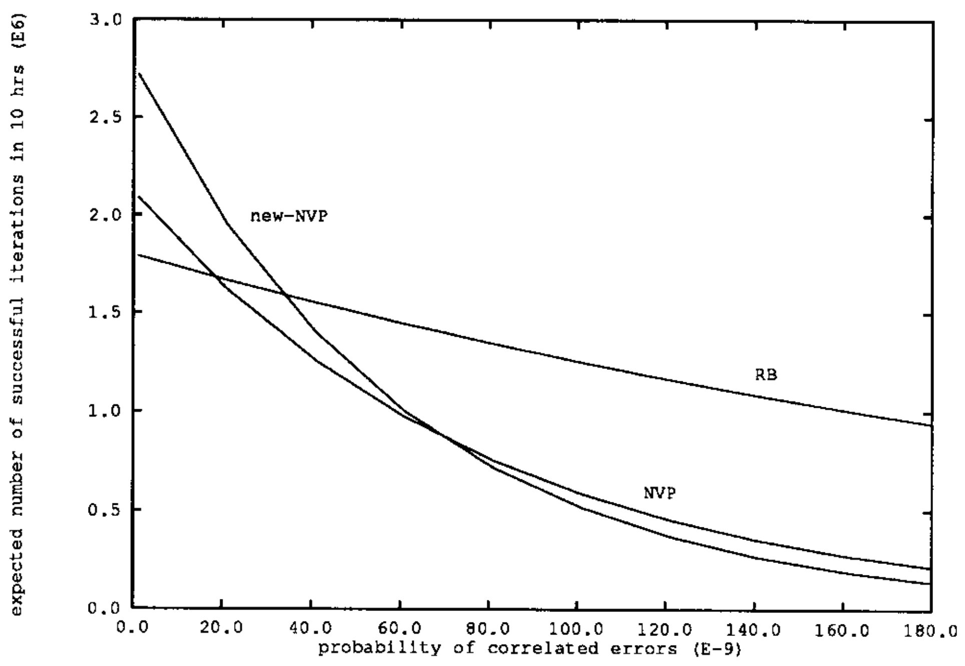


Figure 6.19: Comparison for Uniform Distribution

Figure 6.19 shows the comparison of the expected reward for RB, NVP and new-NVP under the condition that the component execution time are uniformly distributed. The information about the uniform distribution for RB, NVP and new-NVP is listed in Table 6.10, 6.11 and 6.12, respectively. For consistency, we let the mean execution time for components in RB, NVP and new-NVP be the same as those for the case of exponential distribution (results based on this were shown previously in Figure 6.17). Other performance and dependability parameters are the same as those used for the case of exponential distribution.

Table 6.10: Parameters of Uniform Distribution (RB)

Component	Probability distribution	Mean
P	Uniform over $(4.5, 5.5)$	5
S	Uniform over $(7.5, 8.5)$	8
AT	Uniform over $(4.5, 5.5)$	5

Table 6.11: Parameters of Uniform Distribution (NVP)

Component	Probability distribution	Mean
version 1	Uniform over $(4.5, 5.5)$	5
version 2	Uniform over $(5.5, 6.5)$	6
version 3	Uniform over $(7.5, 8.5)$	8
voter	Uniform over $(0.2, 0.8)$	0.5

Table 6.12: Parameters of Uniform Distribution (new-NVP)

Component	Probability distribution	Mean
version 1	Uniform over $(4.5, 5.5)$	5
version 2	Uniform over $(5.5, 6.5)$	6
version 3	Uniform over $(7.5, 8.5)$	8
voter 1	Uniform over $(0.2, 0.8)$	0.5
voter 2	Uniform over $(0.2, 0.8)$	0.5

The comparison shows that, in this setting, RB is inferior to both NVP and new-NVP in the low correlated error probability region, because of the performance penalty from its sequential execution of P and AT. It becomes superior in the higher correlated error probability region due to its dependability merit. Although the new-NVP performs the best for the low correlated error region, its reward is not as high as for the uniform distribution as it is for the exponential distribution. This is due to the fact that the variance of version execution time is much smaller than in the exponential case. The time to completion of two versions is tightly bounded below, so that the improvement is relatively less significant.

6.10 Evaluation for Fault Tolerant Software of Closed Loop Type

We have also evaluated performability measures for fault-tolerant software in applications of the closed loop type. In the evaluation, we assume that the effects of graceful degradation (due to detected and treated errors) will propagate, but will not amplify. We assume that the fault-tolerant software systems have the following disciplines:

- After the first degradation, the quality of service of each successful cycle is reduced and remains as a constant before a subsequent degradation.
- Upon the second degradation, the software system is safely "shut-down." That is, its output is ignored and manual operation takes place; the software system is considered "out of service" from that point on.

Thus, only the cycle which produces an undetectable error and occurs before the second degradation will constitute catastrophic failure.

Since the object systems (RB, NVP and new-NVP) are not changed from the open loop case, the lower layer models defined previously are applicable. Nevertheless, the upper layer, the performability model, needs to be modified in order to reflect the impact of degradation propagation on user perceived benefits. The performance variable $M(t)$ is defined as the number of successful iterations in a mission period t. An iteration after the first degradation and before the second degradation is considered as a partially successful one, and counted as a fraction. For deriving the moment generating function of the performance variable for the closed loop case, we define the following random variables:

$$\begin{aligned}
D(t) &= \text{number of degradations (detected erroneous cycles) in } t, \\
N(t) &= \text{number of failures (undetected erroneous cycles) in } t, \\
K(t) &= \text{total number of cycles in } t, \\
X_0(t) &= \text{number of cycles before the first degradation or a} \\
&\quad\ \text{catastrophic failure,} \\
X_1(t) &= \text{number of cycles after the first degradation and before} \\
&\quad\ \text{the second degradation or a catastrophic failure.}
\end{aligned}$$

And we let w_0 denote the reward impulse associated with a successful cycle before the first degradation, and w_1 denote the reward impulse associated with a successful cycle after the first and before the second degradation.

The service threshold for classifying performance levels is based on $D(t)$ and $N(t)$. The macro-level capability function is:

$$\gamma(u_{\{t_0,t_0+t\}}) \in \begin{cases} A_{fully-proper} & \text{if } D(t) \le 1 \text{ and } N(t) = 0 \\ A_{partially-proper} & \text{if } D(t) > 1 \text{ and } N(t) = 0 \\ A_{improper} & \text{if } N(t) > 0 \end{cases}$$

The reward structure which supports the performance variable is thus defined as follows:

$$Y_{\{t_0,t_0+t\}} = M(t) = \begin{cases} (w_0 - w_1)X_0(t) + w_1\, K(t) & \text{if } \gamma(u) \in A_{fully-proper} \\ w_0\, X_0(t) + w_1\, X_1(t) & \text{if } \gamma(u) \in A_{partially-proper} \\ 0 & \text{if } \gamma(u) \in A_{improper} \end{cases}$$

Since $X_0(t)$, $X_1(t)$, $D(t)$ and $N(t)$ uniquely determines $M(t)$, we denote

$$M = h(x_0, x_1, d, n).$$

Let $Z = e^s$, and we have the moment generating function:

$$\begin{aligned} E[Z^M] &= \sum_{n=0}^{\infty}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) \\ &= \sum_{n=1}^{\infty}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) + \\ &\quad \sum_{n=0}^{0}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n). \end{aligned} \tag{6.19}$$

The first term of the above equation (6.19) can be decomposed, and rewritten by applying the definition of the reward structure. That is,

$$\begin{aligned}
&\sum_{n=1}^{\infty}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) \\
&= \sum_{n=1}^{\infty}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{0} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) + \\
&\quad \sum_{n=1}^{\infty}\sum_{d=1}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) \\
&= \sum_{x_0=0}^{\infty}\sum_{k=x_0+1}^{\infty} (1-p_{dg}-p_{cf})^{x_0}\cdot p_{cf}\cdot P(K(t)=k) + \\
&\quad \sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+1}^{\infty} (1-p_{dg}-p_{cf})^{x_0+x_1-1}\cdot p_{dg}\cdot p_{cf}\cdot P(K(t)=k) \\
&= \frac{p_{cf}}{p_{dg}+p_{cf}} \sum_{k=1}^{\infty} \left(1-(1-p_{dg}-p_{cf})^{k-1}\right) P(K(t)=k) + \\
&\quad \frac{p_{dg}\,p_{cf}}{p_{dg}+p_{cf}} \sum_{k=2}^{\infty} \left(1-k\,(1-p_{dg}-p_{cf})^{k-1}\right) P(K(t)=k) \qquad (6.20)
\end{aligned}$$

The second term of the Eq. (6.19) can also be decomposed and rewritten by applying the definition of the reward structure. That is,

$$\begin{aligned}
&\sum_{n=0}^{0}\sum_{d=0}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) \\
&= \sum_{d=0}^{1}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) + \\
&\quad \sum_{d=2}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=0}^{\infty} Z^{h(x_0,x_1,d,n)}\, P(x_0,x_1,d,n) \\
&= \sum_{d=0}^{1}\sum_{x_0=0}^{\infty}\sum_{k=x_0+d}^{\infty} Z^{h(x_0,x_1,d,n)}(1-p_{dg}-p_{cf})^{x_0}\cdot {p_{dg}}^{d}(1-p_{dg}-p_{cf})^{k-x_0-d}\cdot \\
&\quad P(K(t)=k) + \sum_{d=2}^{\infty}\sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+d-1}^{\infty} Z^{h(x_0,x_1,d,n)}(1-p_{dg}-p_{cf})^{x_0+x_1-1} \\
&\quad {p_{dg}}^{2}\binom{k-x_0-x_1-1}{d-2}{p_{dg}}^{d-2}(1-p_{dg})^{k-x_0-x_1-1-(d-2)}\cdot P(K(t)=k) \\
&= \sum_{d=0}^{1}\sum_{k=d}^{\infty}\sum_{x_0=0}^{k-d} Z^{w_1\,k+(w_0-w_1)\,x_0}(1-p_{dg}-p_{cf})^{k-d}\cdot {p_{dg}}^{d}\cdot P(K(t)=k) + \\
&\quad \sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+d-1}^{\infty} Z^{w_0\,x_0+w_1\,x_1}(1-p_{dg}-p_{cf})^{x_0+x_1-1}{p_{dg}}^{2}\cdot P(K(t)=k)\cdot
\end{aligned}$$

$$\sum_{d=2}^{k-x_0-x_1+1} {p_{dg}}^2 \binom{k-x_0-x_1-1}{d-2} {p_{dg}}^{d-2}(1-p_{dg})^{k-x_0-x_1-1-(d-2)}$$

$$= \sum_{d=0}^{1}\sum_{k=d}^{\infty}\sum_{x_0=0}^{k-d} Z^{w_1\,k+(w_0-w_1)\,x_0}(1-p_{dg}-p_{cf})^{k-d}\cdot {p_{dg}}^{d}\cdot P(K(t)=k) + \sum_{x_0=0}^{\infty}$$
$$\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+1}^{\infty} Z^{w_0\,x_0+w_1\,x_1}(1-p_{dg}-p_{cf})^{x_0+x_1-1}{p_{dg}}^2\cdot P(K(t)=k). \quad (6.21)$$

The moment generating function (Eq. (6.19)) then becomes:

$$E[Z^M] = \frac{p_{cf}}{p_{dg}+p_{cf}}\sum_{k=1}^{\infty}\left(1-(1-p_{dg}-p_{cf})^{k-1}\right)P(K(t)=k) +$$
$$\frac{p_{dg}\,p_{cf}}{p_{dg}+p_{cf}}\sum_{k=2}^{\infty}\left(1-k\,(1-p_{dg}-p_{cf})^{k-1}\right)P(K(t)=k)\ +$$
$$\sum_{d=0}^{1}\sum_{k=d}^{\infty}\sum_{x_0=0}^{k-d} Z^{w_1\,k+(w_0-w_1)\,x_0}(1-p_{dg}-p_{cf})^{k-d}\cdot {p_{dg}}^{d}\cdot P(K(t)=k)\ +$$
$$\sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+1}^{\infty} Z^{w_0\,x_0+w_1\,x_1}$$
$$(1-p_{dg}-p_{cf})^{x_0+x_1-1}{p_{dg}}^2\cdot P(K(t)=k) \quad (6.22)$$

Using the moment generating function Eq. (6.22), we can derive the expected reward:

$$E[M] = \sum_{k=0}^{\infty}(1-p_{dg}-p_{cf})^{k}\cdot P(K(t)=k)\cdot w_0\,k\ +$$
$$\sum_{k=1}^{\infty}(1-p_{dg}-p_{cf})^{k-1}\cdot p_{dg}\cdot P(K(t)=k)\sum_{x_0=0}^{k-1}\left(w_1\,k+(w_0-w_1)\,x_0\right)\ +$$
$$\sum_{x_0=0}^{\infty}\sum_{x_1=1}^{\infty}\sum_{k=x_0+x_1+1}^{\infty}(w_0\,x_0+w_1\,x_1)(1-p_{dg}-p_{cf})^{x_0+x_1-1}{p_{dg}}^2\cdot P(K(t)=k)$$
$$= \sum_{k=0}^{\infty}(1-p_{dg}-p_{cf})^{k}\cdot P(K(t)=k)\cdot w_0\,k\ +$$
$$\sum_{k=1}^{\infty}(1-p_{dg}-p_{cf})^{k-1}\cdot p_{dg}\cdot P(K(t)=k)\cdot\frac{(w_0+w_1)\,k^2-(w_0-w_1)\,k}{2}\ +$$
$$\frac{{p_{dg}}^2}{(1-p_{dg}-p_{cf})}\sum_{k=2}^{\infty}P(K(t)=k)\sum_{x_0=0}^{k-2}(1-p_{dg}-p_{cf})^{x_0}\cdot$$
$$\sum_{x_1=1}^{k-x_0-1}(w_0\,x_0+w_1\,x_1)\cdot(1-p_{dg}-p_{cf})^{x_1}$$

$$= \sum_{k=0}^{\infty}(1-p_{dg}-p_{cf})^k \cdot P(K(t)=k)\cdot w_0\, k \ +$$
$$\sum_{k=1}^{\infty}(1-p_{dg}-p_{cf})^{k-1}\cdot p_{dg}\cdot P(K(t)=k)\cdot \frac{(w_0+w_1)\,k^2-(w_0-w_1)\,k}{2} \ +$$
$$\frac{{p_{dg}}^2}{(1-p_{dg}-p_{cf})}\sum_{k=2}^{\infty}P(K(t)=k)\sum_{i=1}^{6}S_i(k) \tag{6.23}$$

The terms $S_i(k)$ are as follows:

$$\begin{aligned}
S_1(k) &= \frac{-w_0(k-1)\,p^k\,(1-p)+w_0\,p^2(1-p^{k-1})}{(1-p)^3}\\
S_2(k) &= -\frac{w_0\,p^k\,(k^2-3\,k+2)}{2\,(1-p)}\\
S_3(k) &= \frac{w_1\,p\,(1-p^{k-1})}{(1-p)^2}\\
S_4(k) &= -\frac{w_1\,p\,(p^{k+1}-p^2)\,(k+\frac{1}{1-p})}{(1-p)\,(1-\frac{1}{p})}\\
S_5(k) &= \frac{w_1\,p^2\,(k-1)}{(1-p)^2}\\
S_6(k) &= \frac{w_1(k-1)\,p^3\,(1-p)+w_1\,(p^{k+2}-p^3)}{(1-p)^3}
\end{aligned}$$

where $p = e^{\alpha} = 1 - p_{dg} - p_{cf}$.

According to the Central Limit Theorem for Renewal Process, $K(t)$ is approximately normally distributed with the mean t/μ and the variance $t\sigma^2/\mu^3$ for large t. Let $\hat{\mu} = t/\mu$, $\hat{\sigma} = \sqrt{t\sigma^2/\mu^3}$, and $e^{\alpha} = 1 - p_{dg} - p_{cf}$ (as defined above), we have

$$\begin{aligned}
E[M] \ = \ & \int_{-\frac{\hat{\mu}}{\hat{\sigma}}}^{\infty} e^{\alpha(\hat{\sigma}x+\hat{\mu})}\varphi(x)\cdot w_0\,(\hat{\sigma}x+\hat{\mu})\,dx \ + \\
& p_{dg}\int_{-\frac{1-\hat{\mu}}{\hat{\sigma}}}^{\infty} e^{\alpha(\hat{\sigma}x+\hat{\mu}-1)}\varphi(x)\,\frac{(w_0+w_1)\,(\hat{\sigma}x+\hat{\mu})^2-(w_0-w_1)\,(\hat{\sigma}x+\hat{\mu})}{2}\,dx \\
& + \frac{{p_{dg}}^2}{(1-p_{dg}-p_{cf})}\int_{-\frac{2-\hat{\mu}}{\hat{\sigma}}}^{\infty}\varphi(x)\sum_{i=1}^{6}S_i(\hat{\sigma}x+\hat{\mu})\,dx
\end{aligned} \tag{6.24}$$

Performability measures are evaluated for RB, NVP and new-NVP schemes with the parameters listed in Table 6.13, 6.14 and 6.15, respectively. Figures 6.20

and 6.21 show the evaluation results for an open loop application and a closed loop application, respectively, based on the parameter assignments. For the open loop application, the expected reward is evaluated using the moment generating function defined in Section 6.5.3. Each successful iteration is always associated with a reward value w_0 which equals unity, because the effects of degradation do not propagate. For the closed loop case, the expected reward is evaluated using Eq. (6.24). To take into account the effects of degradation propagation, each successful iteration before first degradation is assigned a reward value w_0 which equals unity; whereas each successful one after the first degradation and before the second degradation is given a reward value w_1 equal to 0.8.

Table 6.13: Assignments of Parameters for RB

Parameter	Value
λ_p	$\frac{1}{5}$
λ_s	$\frac{1}{8}$
λ_t	$\frac{1}{5}$
q_{ps}	variable
q_{pt}	variable
q_{st}	variable
q_{pst}	10^{-10}
q_p	10^{-6}
q_s	10^{-6}
q_t	10^{-6}
t	$3.6\ 10^7$ (ms)
τ	100 (ms)

Compared with the open loop case, RB receives a noticeably poorer reward for the closed loop. Eq.'s (6.2), (6.13) and (6.15) show that the probability of

Table 6.14: Assignments of Parameters for NVP

Parameter	Value
λ_1	$\frac{1}{5}$
λ_2	$\frac{1}{6}$
λ_3	$\frac{1}{8}$
λ_4	$\frac{1}{0.5}$
q_{2v}	variable
q_{3v}	10^{-10}
q_{iv}	10^{-6}
q_{d_1}	10^{-9}
q_{d_2}	10^{-9}
t	$3.6\ 10^7$ (ms)
τ	100 (ms)

Table 6.15: Assignments of Parameters for new-NVP

Parameter	Value
λ_1	$\frac{1}{5}$
λ_2	$\frac{1}{6}$
λ_3	$\frac{1}{8}$
λ_4	$\frac{1}{0.5}$
λ_5	$\frac{1}{0.5}$
q_{2v}	variable
q_{3v}	10^{-10}
q_{iv}	10^{-6}
q_{d1}	10^{-9}
q_{d2}	10^{-9}
t	$3.6\ 10^7$ (ms)
τ	100 (ms)

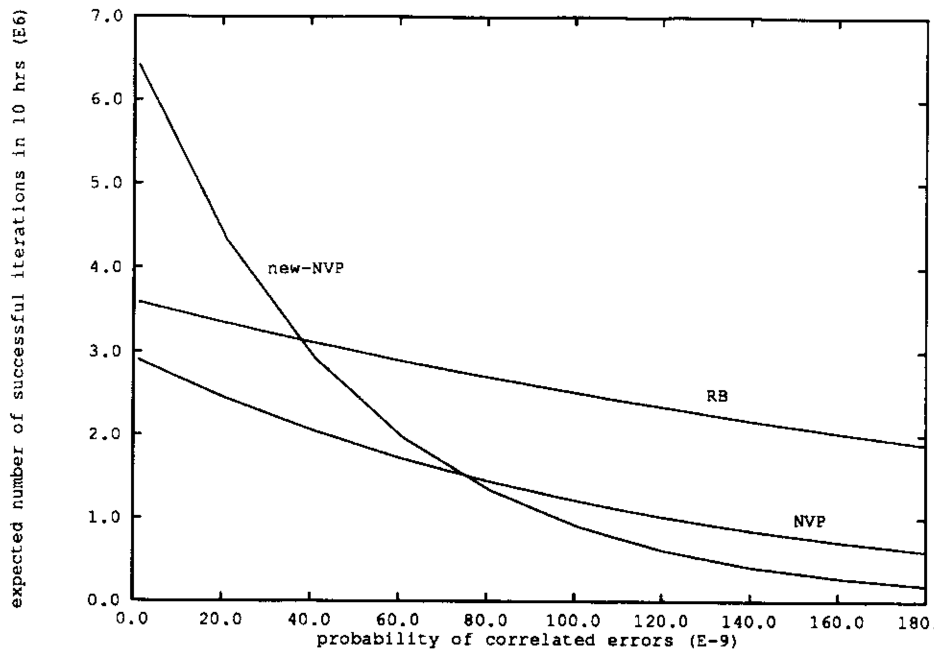


Figure 6.20: Comparison for Open Loop Case

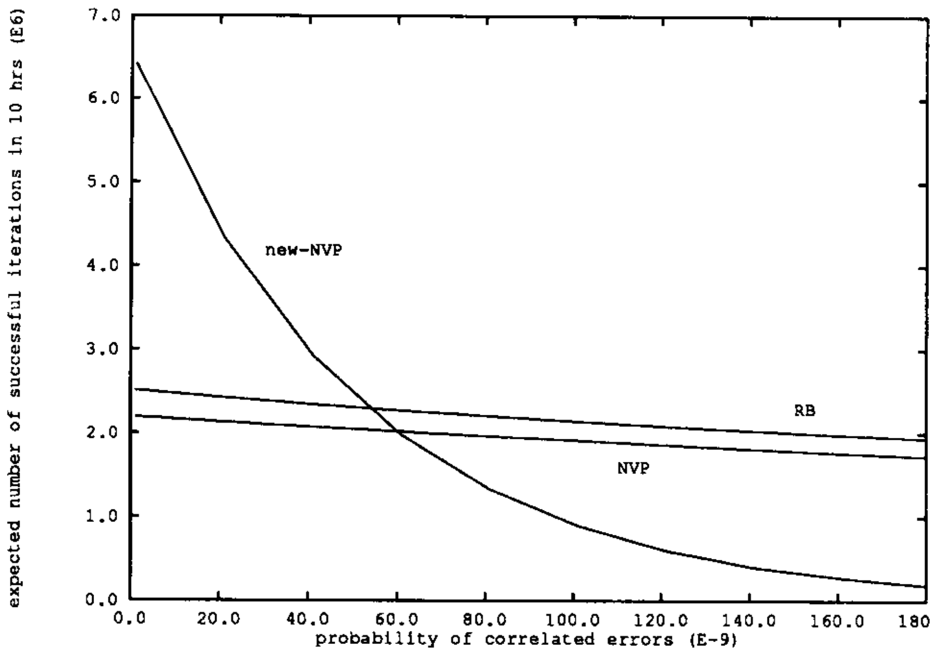


Figure 6.21: Comparison for Close Loop Case

independent errors in a decision function (an acceptance test or a voter) dominates the likelihood of degradation. When an acceptance test is designed under the temptation of full coverage, its error probability can be comparable with that associated with an alternative. On the other hand, since a voter in NVP or new-NVP is application independent and easier to verify, it is legitimate to assume that the error probability of a voter is lower than that of a version. Therefore, the RB scheme is more vulnerable to service degradation. Since the degradation has more significant effects on the expected reward for a closed loop application than for an open loop application, the change in the effectiveness of RB is a reasonable result.

The results also reveal that the expected reward for RB and NVP is relatively insensitive to the correlated error probability. This is due to the fact that the system will no longer be active after the second degradation. Under this operational discipline, the effect of correlated errors, which causes the loss of a mission, is suppressed after being twice degraded.

The new-NVP behavior is rather similar for the open loop and closed loop cases. The factors associated with this phenomenon are:

1. The high performance of new-NVP results in a low probability of timing errors.

2. The dependable voting scheme of new-NVP results in a low probability of unnecessary result suppression.

Thus, new-NVP has a lower probability of degradation, which makes its expected reward less sensitive to the effects of degradation propagation.

CHAPTER 7

Conclusions and Future Work

In this research, we have demonstrated the applicability of performability concepts to real-time software and the approaches to model construction for capturing the characteristics of software behavior. The mathematically based framework is able to address a broad spectrum of software issues. Evaluation results show that performability modeling can provide answers that form the basis for objective, quantitatively derived, decisions for design and operational aspects of real-time systems. The results go beyond what could be expected from intuitive insights or unaided reasoning.

7.1 Reward Structure

Real-time software incorporating defensive programming or fault tolerance strategies may exhibit both gracefully and non-gracefully degradable performance. Aimed at capturing the characteristics of real-time software, we have explored the methodology to evaluate the unified measures via distinction. The distinction is achieved by defining an integrated relationship of the top-down and bottom-up capability functions in a reward structure. The top-down capability function $\gamma(u)$ summarizes the software behavior at a macro level and qualitatively identifies if the service threshold can be met by the system under question. Conditioned on the macro-level classification, a bottom-up capability function $\tilde{\gamma}(u)$ quantitatively measures the quality of service at a micro-level. While the stronger probabilistic measures, such as PDF, at a micro-level are often difficult to evaluate by analytical methods, these at a macro-level are in general analytically attainable. The stronger probabilistic descriptions at the macro-level screen out systems (or designs) which violate the threshold, thus meaningful quantitative comparisons or tradeoffs can be effectively performed by employing the weaker measures such as expected reward at the micro-level. In summary, this integration is necessary in order to provide operationally meaningful and mathematically realizable performability measures. Further investigation is needed to formalize and rigorously establish this relationship. We also notice that, since this approach permits distinctive yet coherent formulations of system effectiveness measures to across the boundaries of the classes, further combination of system attributes becomes possible. For example, for the class of improper service (or non-gracefully degraded service) safety notions can be applied to refine the measure of "loss."

7.2 Hierarchical Approach

A hierarchical approach is used to construct performability models for assessing the effectiveness of software fault tolerance techniques. Lower layer models describe the software failure behavior and performance characteristics of fault tolerance systems; the lower layer evaluates the local performance and dependability measures [1] with the consideration of their dependencies. The results are supplied to the upper layer. The upper layer accomplishes an integrated evalua-

[1]We use the term "local measures" to refer to the individual entities in a total system, such as the failure probability and mean execution time of a single program iteration.

tion by employing these local measures as key parameters of the performability model.

The hierarchical approach enables model validation by use of separate dependability and performance field data. The use of such separate dependability and performance data is necessary because there may not be sources of field performability data. Model maintainability and extensibility are enhanced by the hierarchical approach, which allows modifications to be made at a specific level rather than throughout the model as a design evolves or the model is applied to a different system. This logical modularity facilitates concurrent model changes and software maintenance.

7.3 How the Reward Structure and Hierarchical Approach relate to SANs

The reward structure proposed in Chapter 5 and applied in Chapter 6 can also be realized by stochastic activity networks. SAN has the capability of performing "marking dependent" reward assignment. By exploiting this capability, reward evaluation can be based on a collective view of the state trajectory and be governed by the system's qualification at the macro level.

In the SAN realization (Chapter 4), the model construction is based on the architecture of a total system (ts-based); when we use straight analytical methods for the evaluation of fault-tolerant software, the hierarchical approach is based on the major attributes of the system (attributes-based). Figure 7.1 contrasts the two approaches.

Accordingly, the SAN realization results in a modularity in a physical sense,

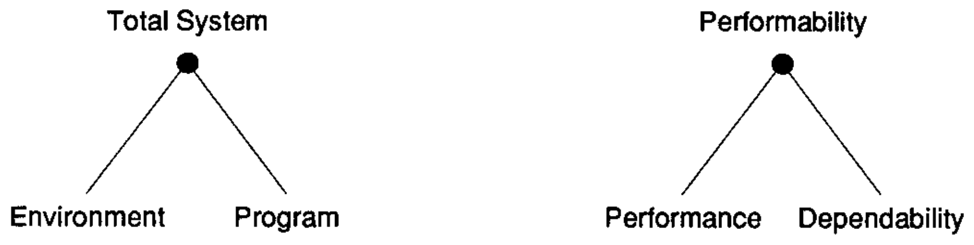


Figure 7.1: Two Types of Modularity

while the hierarchical approach results in a modularity in a logical sense. Both types of modularity exhibit advantages and disadvantages. The physical modularity allows a more natural representation. It is able to accommodate high complexity especially when software evaluation tools with a simulation option are available. Models based on physical modularity may directly provide insight on the interaction between the object software and its operational environment. The logical modularity usually entails a simpler representation. Models based on the logical modularity may directly support parametric studies and provide to us insight about the contributions of performance and dependability attributes to the overall service quality and dependencies between these attributes. Logical modularity favors local modifications and thus enhances model maintainability.

The disadvantages of using the physical modularity are 1) the model construction process is error-prone due to complex representation, and modifications usually involve multiple sub-models so that maintenance is difficult, and 2) there is no direct access to the contribution of performance and dependability to the overall effectiveness. The disadvantages of using the logical modularity are: 1) applicability is limited by the complexity of the object system, and 2) difficulty exists for explicitly incorporating environmental attributes.

Our future direction is to explore an approach to integrate the concepts of physical and logical modularities.

7.4 Summary of Future Work

In summary, future research issues include:

1. To formalize the reward structure for software performability evaluation.
2. To generalize the hierarchical methods for model construction and solution for incorporating more environmental attributes.
3. To explore the use of existing graphical tools, such as those based on stochastic Petri Nets, to evaluate more complicated systems.

APPENDIX A

Glossary of Notation

$A_{improper}$	performance level set of improper service
A_{proper}	performance level set of proper service
CF	the event of catastrophic failure
CF^c	the absence of catastrophic failure
$\hat{D}$	maximum degree of degradation allowed by the service threshold
d	number of degradation
E	environment sub-model
$f(y)$	density function of program iteration time
$g(y)$	density function of program iteration time with deadline consideration
$I_{\{t_0,t_0+t\}}$	collective indicator vector for the state trajectory in $[t_0, t_0+t]$
$K(t)$	number of total (program) iterations in time t
$M(t)$	number of successful iterations perceived by the user in time t
NVP	N-Version Programming
new-NVP	an alternative approach to N-Version Programming

P	program submodel
PDF	probability distribution function
pdf	probability density function
p_{cf}	probability of an undetected erroneous iteration
p_{dg}	probability of a degraded iteration
p_{sd}	probability of a detected erroneous iteration
p_{tt}	probability that an iteration results in real-time deadline violation
q	probability of errors in a non-fault-tolerant program
q_d	probability of independent errors in the voter
q_{iv}	probability of independent errors the i^{th} version
q_{ps}	probability of related errors between P and S
q_{pt}	probability of related errors between P and AT
q_{pst}	probability of related errors between P, S and AT
q_p	probability of independent errors in primary
q_s	probability of independent errors in secondary
q_{st}	probability of related errors between S and AT
q_t	probability of independent errors in acceptance test
q_{2v}	probability of related errors between two versions
q_{3v}	probability of related errors between three versions

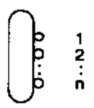
RB	Recovery Blocks
SP	Software Performability Model
t	length of mission period
TS	Total System
u	state trajectory
$u_{\{t_0,t_0+t\}}$	a collective view of the state trajectory over a mission period $[t_0, t_0+t]$
V	a generic performance variable
$Y_{\{t_0,t_0+t\}}$	performance variable summarizing the software behavior over $[t_0, t_0+t]$
y	program iteration time
$\delta(y-\tau)$	unit impulse function
γ	"Top-down" capability function
$\tilde{\gamma}$	reward-based capability function
λ	execution rate of a non-fault-tolerant program
λ_p	execution rate of the primary
λ_s	execution rate of the secondary
λ_t	execution rate of the acceptance test
λ_1	execution rate of the first version
λ_2	execution rate of the second version
λ_3	execution rate of the third version

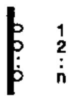
λ_4	execution rate of the voter
λ_5	execution rate of the second voter (in new-NVP)
μ	mean of iteration time
σ^2	variance of iteration time
τ	real-time deadline for an iteration

APPENDIX B

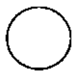
SAN Structure

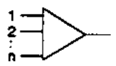
A SAN consists of the following primitive elements (based on [6]):

i) *activities*, which are of two kinds: *timed* activities and *instantaneous* activities. Each activity has a finite set of *cases* (at least one). A timed activity with n cases is depicted as 

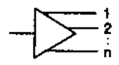
. An instantaneous activity with n cases is shown as 

.

ii) *places*, depicted as  .

iii) *input gates*, which have a finite set of inputs and one output. An input gate with n inputs is depicted as 

. Each such input gate is associated with a n-nary computable predicate and a n-ary computable partial function on the set of natural numbers called the *enabling predicate* and the *input function*, respectively. The input function is defined for all values for which the enabling predicate is true.

iv) *output gates*, which have a finite set of outputs and one input. An output gate with n outputs is depicted as 

. Each such output

gate is associated with a n-ary computable function on the set of natural numbers called the *output function.*

Structurally, an activity network is an interconnection of a finite number of primitives, subject to the following connection rules:

1. Each input of an input gate is connected to a unique place and the output of an input gate is connected to a single activity.
2. Different input gates of an activity are connected to different places.
3. Each output of an output gate is connected to a unique place and the input of an output gate is connected to a single activity (via some cases).
4. Different output gates of an activity for a case are connected to different places.
5. Each place and activity is connected to some input gate or output gate.

In order to increase the understandability of SANs, the following conventions are used in their graphical representation:

a) Activities with one case are shown without any case.

b) An input gate with one input, enabling predicate $e(x) : x \geq 1$, and input function f such that $f(x) = x - 1$, is shown as a directed line from its input to its output.

c) An output gate with one output and output function f such that $f(x) = x + 1$ is shown as a directed line from its input to its output.

d) An input gate with one input, enabling predicate $e(x)$: 1 (always enabled), and an identity input function is shown as a direct line to its output without any input.

```
            exit(0);
        fscanf(f1, "%lf", &prob_suc);
        fscanf(f1, "%lf", &prob_fail);

        reward = mission_rwd(alpha, prob_suc, prob_fail);

        fprintf(fout, "%12.5lf        %15.8lf\n", alpha, reward);
        fflush(fout);
    }
}
double
mission_rwd(alpha, prob_suc, prob_fail)
    double          alpha, prob_suc, prob_fail;
{
    int             i, k;
    double          sum, accltd, pois, prev_pois;
    double          checksum;

    k = 0;
    sum = 0.0;
    prev_pois = -1.0;
    while (1) {
        accltd = 0.0;
        pois = poisson(alpha, TAU, k);
        fflush(stdout);

        if ((pois < prev_pois) && (pois < 0.0000000000000001)) {
            printf("poisson = %20.16lf, k= %d\n", pois, k);
            return (sum);
        } else {
            prev_pois = pois;
            for (i = 0; i <= k; i++) {
            accltd += ((double) (k - i) * oc_suc +
      (double) i * oc_fail) * comb(k, i)
      * pow(prob_suc, (double) (k - i)) *
      pow(prob_fail, (double) i);
            }
            accltd = accltd * pois;
            sum += accltd;
            k += 1;
        }
    }
}
double
poisson(alpha, tau, k)
    double          alpha, tau;
    int             k;
{
    int             j, m;
    double          prod;

    prod = 1.0;
    for (j = 1; j <= k; j++) {
        prod = prod * (alpha * tau) / (double) j;
    }

    m = (int) (alpha * tau) / 10;
    for (j = 1; j <= m; j++)
        prod *= exp(-10.0);
    m = (int) (alpha * tau) % 10;
    prod *= exp(-(double) m);
    return (prod);
```

C.2 Program for Solving a Markov Model for Steady State

```
/*
 * program title:  steady
 *
 * author:  Ann T. Tai
 *
 * summary:  This program takes a set of balance equations as its
 *           input.  It solves the steady state occupancy
 *           probabilities using the methods of matrix inversion
 */          and error compensation.

#include        <math.h>
#include        <stdio.h>

#define SIZE        7

main(argc, argv)
    int             argc;
    char           *argv[];
{
    double          matrix[SIZE][2 * SIZE];
    double          a[SIZE][2 * SIZE];
    double          b[SIZE];
    double          x[SIZE];
    double          fail_prob;
    double          beta, e, cvg, alpha;
    double          delta;
    int             mode;
    int             iteration;
    double          xcoord;
    FILE           *plotfile;

    plotfile = fopen("plot.dat", "w");
    if (plotfile == (FILE *) NULL) {
        printf("Can't open plot.dat for write\n");
        exit(-1);
    }
    scanf("%lf", &alpha);
    scanf("%lf", &e);
    scanf("%lf", &cvg);
    scanf("%lf", &beta);
    scanf("%d", &mode);
    scanf("%lf", &delta);
    scanf("%d", &iteration);

    switch (mode) {
    case 1:
        xcoord = alpha;
        break;
    case 2:
        xcoord = e;
        break;
    case 3:
        xcoord = cvg;
        break;
    case 4:
        xcoord = beta;
        break;
    }
    printf("%10.4lf", alpha);
    printf("%10.4lf", e);
    printf("%10.4lf", cvg);
    printf("%10.4lf", beta);
    double          r[SIZE];
    double          e[SIZE];

    /* the initial solution */
    for (i = 0; i < SIZE; i++) {
        x[i] = 0.0;
        for (j = SIZE; j < 2 * SIZE; j++)
            x[i] = x[i] + inv[i][j] * b[j - SIZE];
    }
    found = count = 0;
    while (!found) {
        for (i = 0; i < SIZE; i++) {
            temp = 0.0;
            for (j = 0; j < SIZE; j++) {
                temp = temp + a[i][j] * x[j];
                r[i] = b[i] - temp;
            }
        }
        for (i = 0; i < SIZE; i++) {
            e[i] = 0.0;
            for (j = SIZE; j < 2 * SIZE; j++) {
                e[i] = e[i] + inv[i][j] * r[j - SIZE];
            }
        }
        norme = 0.0;
        for (i = 0; i < SIZE; i++) {
            x[i] = x[i] + e[i];
            norme = norme + e[i];
        }
        count = count + 1;
        if ((norme < 0.005) || (count == 20))
            found = 1;
    }
}
double
failure_prob(x)
    double          x[SIZE];
{
    double          p;
    int             i;

    p = 0;
    for (i = 0; i < SIZE; i++)
        if (i % 2 == 1)
            p += x[i];

    return (p);
}
read_matrix(a, b, matrix)
    double          a[SIZE][2 * SIZE];
    double          b[SIZE];
    double          matrix[SIZE][2 * SIZE];
{
    int             i, j;
    double          v;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            if (scanf("%lf", &v) != 1) {
                return (-1);
```

APPENDIX D

Sample Mathematica Program and Output

D.1 Mathematica Code Computing the Expected Reward of NVP

```
(*********************************************)
(*              Fault Model                  *)
(*********************************************)
pcf := (3 qiv^2 (1 - qiv) + qiv^3) qd2 + 3 q2v (1 - qd1) +
q3v (1 - qd1);
psd := (1 - 3 qiv^2 (1 - qiv)) qd1 + (3 qiv^2 (1 - qiv) + qiv^3)
(1 - qd2);
qiv = 0.0001;
qd1 = 10^-9;
qd2 = 10^-9;
q2v = 10^-9;
q3v = 10^-10;
(*********************************************)
(*           Performance Model               *)
(*********************************************)
<< Laplace.m;
H := Expand[(1 - Exp[-lambda1 y]) (1 - Exp[-lambda2 y])
(1 - Exp[-lambda3 y])];
h := Simplify[D[H, y]];
(* Print["h(y) = ", h]; *)
lh := Laplace[h, y, s];
(* Print["H*(s) = ", lh]; *)
g := lambda4 Exp[-lambda4 y];
lg := Laplace[g, y, s];
Print["G*(s) = ", lg];
F := Expand[lh lg];
(* Print["F*(s) = ", F]; *)
f = Ilaplace[Apart[F,s], s, y];
(* Print["f(y) = ", f]; *)
ptt = N[1 - Integrate[f, {y, 0, tau}]];
pdg = ptt + psd;
lambda1 = 1/5; lambda2 = 1/6; lambda3 = 1/8;
lambda4 = 2;
tau = 30;
first := Integrate[Expand[y f], {y, 0, tau}] + tau ptt;
Print["first = ", first];
second := Integrate[Expand[y^2 f], {y, 0, tau}] + tau^2 ptt;
```

D.2 Mathematica Output for the Expected Reward of NVP

```
Mathematica (DEC RISC) 1.2 (January 24, 1990) [With pre-loaded data]
by S. Wolfram, D. Grayson, R. Maeder, H. Cejtin,
   S. Omohundro, D. Ballman and J. Keiper
with I. Rivin and D. Withoff
Copyright 1988,1989,1990 Wolfram Research Inc.
 -- Terminal graphics initialized --

In[1]:=
In[2]:=
In[3]:=
In[4]:=
In[5]:=
In[6]:=
In[7]:=
In[8]:=
In[9]:=
In[10]:=
In[11]:=
In[12]:=
In[13]:=
In[14]:=
In[15]:=
In[16]:=
In[17]:=
In[18]:=
In[19]:=
In[20]:=
In[21]:=
In[22]:=  lambda4
G*(s) = -----------
        lambda4 + s

In[23]:=
In[24]:=
In[25]:=
In[26]:=
In[27]:=
In[28]:=
In[29]:=
In[30]:=
In[31]:=
In[32]:=
In[33]:=          3974223749         453600       21600
first = 13.3486 + --------------- - ----------- + ------- +
                              60            59/4        11
                  24119511570 E      10679 E       539 E

      34400        11232      350     432      608
>     --------- + --------- - ---- - ----- - --------
           39/4         35/4     6        5        15/4
```

```
pdg = 0.0349152
mu = 12.0435
sigma2 = 48.877
                        6
muhead = 2.98917 10
sigmahead = 1003.63
                        6
reward = 1.99669 10
Accuracy[reward] = 10
                                                                       -8
The iteration for co-related errors in two versions = 6.1 10
              -7
pcf = 1.831 10
ptt = 0.0349152
               -8
psd = 3.0998 10
pdg = 0.0349152
mu = 12.0435
sigma2 = 48.877
                        6
muhead = 2.98917 10
sigmahead = 1003.63
                        6
reward = 1.66886 10
Accuracy[reward] = 10
                                                                       -8
The iteration for co-related errors in two versions = 8.1 10
              -7
pcf = 2.431 10
ptt = 0.0349152
               -8
psd = 3.0998 10
pdg = 0.0349152
mu = 12.0435
sigma2 = 48.877
                        6
muhead = 2.98917 10
sigmahead = 1003.63
                        6
reward = 1.39485 10
Accuracy[reward] = 10
                                                                        -7
The iteration for co-related errors in two versions = 1.01 10
              -7
pcf = 3.031 10
ptt = 0.0349152
               -8
psd = 3.0998 10
pdg = 0.0349152
mu = 12.0435
sigma2 = 48.877
                        6
muhead = 2.98917 10
```

```
ptt = 0.0349152
                   -8
psd = 3.0998 10
pdg = 0.0349152
mu = 12.0435
sigma2 = 48.877
                       6
muhead = 2.98917 10
sigmahead = 1003.63
reward = 568950.
Accuracy[reward] = 10

In[38]:=
F1tau := Integrate[f1, {x, 0, tau}];
first1 := Integrate[Expand[x f1], {x, 0, tau}];
second1 := Integrate[Expand[x^2 f1], {x, 0, tau}];
(* Print["w = ", w]; *)
Print["w is done ..."];
(* w is the PDF of (Y2+Yv2) *)
W = Integrate[w, {x, 0, y}];
(* Print["W = ", W]; *)
Print["W is done ..."];
(* V is the PDF of Y3 *)
V = (1 - Exp[-lambda1 y]) (1 - Exp[-lambda2 y]) (1 - Exp[-lambda3 y]);
(* G2 is the PDF of MAX((Y2+Y2v) and Y3) *)
G2 = Expand[W V];
Clear[W, V, lw, w];
(* Print["G2 = ", G2]; *)
Print["G2 is done ..."];
(* g2 is the pdf of MAX((Y2+Y2v) and Y3) *)
g2 = Simplify[Collect[Expand[D[G2, y]], y]];
(* Print["g2 = ", g2]; *)
Print["g2 is done ..."];
(* lg2 is the LTx of MAX((Y2+Y2v) and Y3) *)
lg2 = Laplace[g2, y, s];
(* Print["lg2 = ", lg2]; *)
Print["lg2 is done ..."];
(* h2 is the pdf of Yv3 *)
h2 = lambda5 Exp[-lambda5 y];
(* lh2 is the LTx of Yv3 *)
lh2 = Laplace[h2, y, s];
(* Print["lh2 = ", lh2]; *)
Print["lh2 is done ..."];
Clear[G2, g2, h2];
(* lf2 is the LTx of (MAX((Y2+Yv2) and Y3) + Yv3) *)
lf2 = Collect[Expand[lg2 lh2], s];
Clear[lg2, lh2];
(* F is the LTx of the renewal cycle time *)
Print["Evaluating partial fraction form of F*(s) ..."];
F2 = Apart[lf2, s];
Clear[lf2];
(* Print["Partial fraction form of F2*(s) = ", F2]; *)
f2 = Ilaplace[F2, s, y];
Print["f2(y) is done ..."];
(* Print["f2(y) = ", f2]; *)
Clear[F2];
F2tau := Integrate[f2, {y, 0, tau}];
first2 := Integrate[Expand[y f2], {y, 0, tau}];
second2 := Integrate[Expand[y^2 f2], {y, 0, tau}];
Ptimeout := 1 - (p2 F1tau + (1 - p2) F2tau); (* = *)
first := p2 first1 + (1 - p2) first2 + tau ptt; (* = *)
(* Print["first = ", N[first]]; *)
second := p2 second1 + (1 - p2) second2 + tau^2 ptt; (* = *)
(* Print["second = ", N[second]]; *)
t = 3.6 10^6;
lambda1 = 1/0.5; lambda2 = 1/0.6;
```

D.4 Mathematica Output for the Expected Reward of New-NVP

```
Mathematica (DEC RISC) 1.2 (January 24, 1990) [With pre-loaded data]
by S. Wolfram, D. Grayson, R. Maeder, H. Cejtin,
   S. Omohundro, D. Ballman and J. Keiper
with I. Rivin and D. Withoff
Copyright 1988,1989,1990 Wolfram Research Inc.
 -- Terminal graphics initialized --

In[1]:=
In[2]:=
In[3]:=
In[4]:=
In[5]:=
In[6]:=
In[7]:=
In[8]:=
In[9]:=
In[10]:=
In[11]:=
In[12]:=
In[13]:=
In[14]:=
In[15]:=
In[16]:=
In[17]:=
In[18]:=
In[19]:=
In[20]:=
In[21]:=
In[22]:=
In[23]:=
In[24]:=
In[25]:=
In[26]:= g1 is done ...

In[27]:=
In[28]:=
In[29]:=
In[30]:= lg1 is done ...

In[31]:=
In[32]:=
In[33]:=
In[34]:=
In[35]:=
In[36]:= lh1 is done ...

In[37]:=
In[38]:=
In[39]:=
In[40]:=
In[86]:=
In[87]:=
In[88]:=
In[89]:=
In[90]:=
In[91]:=
In[92]:=
In[93]:=
In[94]:=
In[95]:=
In[96]:=
In[97]:=
In[98]:=
In[99]:=
In[100]:=                                                        -9
The iteration for co-related erorrs in two versions = 1. 10
p = 0.9998
p2 = 0.9998
ptt = 0.000265981
                 -8
psd = 2.99982 10
pdg = 0.000266011
                -9
pcf = 3.1002 10
mu = 0.549892
sigma2 = 0.141776
                   6
muhead = 6.54674 10
sigmahead = 1752.01
                  6
reward = 6.4135 10
Accuracy[reward] = 9
                                                               -8
The iteration for co-related erorrs in two versions = 2.1 10
p = 0.9998
p2 = 0.9998
ptt = 0.000265983
                 -8
psd = 2.99982 10
pdg = 0.000266013
                 -8
pcf = 6.31002 10
mu = 0.549892
sigma2 = 0.141776
                   6
muhead = 6.54674 10
sigmahead = 1752.01
                   6
reward = 4.33014 10
Accuracy[reward] = 9
                                                               -8
The iteration for co-related erorrs in two versions = 4.1 10
The iteration for co-related erorrs in two versions = 1.01 10
p = 0.9998
p2 = 0.9998
ptt = 0.000265988
                  -8
psd = 2.99982 10
pdg = 0.000266018
                -7
pcf = 3.031 10
mu = 0.549892
sigma2 = 0.141776
                     6
muhead = 6.54674 10
sigmahead = 1752.01
reward = 899769.
Accuracy[reward] = 10
                                                               -7
The iteration for co-related erorrs in two versions = 1.21 10
p = 0.9998
p2 = 0.9998
ptt = 0.00026599
                  -8
psd = 2.99982 10
pdg = 0.00026602
                -7
pcf = 3.631 10
mu = 0.549892
sigma2 = 0.141776
                     6
muhead = 6.54674 10
sigmahead = 1752.01
reward = 607488.
Accuracy[reward] = 10
                                                               -7
The iteration for co-related erorrs in two versions = 1.41 10
p = 0.9998
p2 = 0.9998
ptt = 0.000265991
                  -8
psd = 2.99982 10
pdg = 0.000266021
                -7
pcf = 4.231 10
mu = 0.549892
sigma2 = 0.141776
                     6
muhead = 6.54674 10
sigmahead = 1752.01
reward = 410152.
Accuracy[reward] = 10
                                                               -7
The iteration for co-related erorrs in two versions = 1.61 10
p = 0.9998
```

REFERENCES

[1] M. J. Pitarys, "Modular embedded computer software for advanced avionics system," in *Proc. NAECON*, pp. 628–633, May 1990.

[2] A. Avižienis and J.-C. Laprie, "Dependable computing: From concepts to design diversity," *Proceedings of the IEEE*, vol. 74, pp. 629–638, May 1986.

[3] J. F. Meyer, "On evaluating the performability of degradable computing systems," in *Proc. Int. Symposium on Fault-Tolerant Computing*, (Toulouse, France), pp. 44–49, June 1978.

[4] J. F. Meyer, "On evaluating the performability of degradable computing systems," *IEEE Transactions on Computers*, vol. C-29, pp. 720–731, Aug. 1980.

[5] J. F. Meyer, "Performability evaluation of the SIFT computer," *IEEE Transactions on Computers*, vol. C-29, pp. 501–509, June 1980.

[6] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symposium*, (Austin, TX), Dec. 1984.

[7] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proc. Int. Workshop on Timed Petri Nets*, (Torino, Italy), pp. 106–115, July 1985.

[8] W. H. Sanders, "Construction and solution of performability models based on stochastic activity networks," computing research laboratory technical report crl tr-9-88, The University of Michigan, Ann Arbor, MI, Aug. 1988.

[9] W. H. Sanders and J. F. Meyer, "Metasan: A performability evaluation tool based on stochastic activity networks," in *Proc. ACM-IEEE Comp. Soc. 1986 Fall Joint Computer Conference*, (Dallas, TX), Nov. 1986.

[10] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Transactions on Computer*, vol. C-29, pp. 540–547, Aug. 1980.

[11] X. Castillo and D. P. Siewiorek, "Workload, performance, and reliability of digital computing systems," in *Proc. Int. Symposium on Fault-Tolerant Computing*, (Portland, ME), pp. 84–89, June 1981.

[22] M. C. Hsueh and B. K. Iyer, "A measurement-based performability model for a multiprocessor system," in *Computer Performance and Reliability* (G. Iazeolla *et al.*, eds.), pp. 337–351, Elseview Science Publishers B. V., North-Holland, 1988.

[23] A. Tai, J. Meyer, and H. Hecht, "A performability model for real-time software," in *Proc. First International Workshop on Performability Modeling of Computer and Communication Systems*, (Enschede, Holland), Feb. 1991.

[24] H. Hecht, "Fault tolerant software for real-time applications," *Computing Surveys*, vol. 8, no. 4, pp. 391–407, 1976.

[25] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," in *Proc. International Working Conference on Dependable Computing for Critical Applications*, (Santa Barbara, CA), pp. 87–94, Aug. 1989.

[26] R. A. Howard, *Dynamic Probabilistic Systems Vol. II: Semi-Markov and Decision Processes.* New York: Wiley, 1971.

[27] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evolution," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1502–1510, December 1985.

[28] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220–232, June 1975.

[29] K. H. Kim and J. C. Yoon, "Approaches to implementation of a repairable distributed recovery block scheme," in *Digest of 18th Annual International Symposium on Fault-Tolerant Computing*, (Tokyo, Japan), pp. 50–55, June 1988.

[30] A. Avižienis and L. Chen, "On the implementation of N-Version Programming for software fault-tolerance during program execution," in *Proceedings of COMPSAC-77*, pp. 149–155, 1977.

[31] A. Avižienis and J. P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer Magazine*, vol. 17, pp. 67–80, August 1984.

[32] A. Avižienis, "The N-Version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1491–1501, December 1985.

[45] P. G. Bishop and F. D. Pullen, "Error masking: A source of failure dependency in multiversion programs," in *Proc. 1st IFIP Working Conference on Dependable Computing for Critical Applications*, (Santa Barbara, CA), pp. 25–32, Aug. 1989.

[46] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1511–1517, Dec. 1985.

[47] J. C. Knight and N. G. Leverson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 96–109, January 1986.

[48] F. Saglietti and W. Ehrenberger, "Software diversity - some considerations about its benefits and its limitations," in *Proceedings of the Fifth IFAC Workshop, SAFECOMP'86*, (Sarlat, France), pp. 27–34, Oct. 1986.

[49] A. Avižienis, M. R. Lyu, and W. Schuetz, "In search of effective diversity: A six-language study of fault-tolerant flight control software," in *Proceedings 18th Annual International Symposium on Fault-Tolerant Computing*, (Tokyo, Japan), pp. 15–22, June 1988.

[50] J. P. J. Kelly and S. C. Murphy, "Achieving dependability throughout the development process: A distributed software experiment," *IEEE Transactions on Software Engineering*, vol. SE-16, pp. 153–165, February 1990.

[51] D. R. Cox, *Renewal Theory*. London: Methuen & Co. Ltd, 1955.

[52] S. Karlin and H. Taylor, *A First Course in Stochastic Processes*. New York: Academic Press, 1975.