SOFTWARE RELIABILITY MODELS: A SURVEY AND AN ANALYSIS

BY

QIAOLAN WAN

Bachelor of Science
Northwest Agricultural University
Xi'an, China
1993

Master of Science
The Graduate School of Chinese Academy of
Agricultural Science
Beijing, China
1995

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2000

SOFTWARE RELIABILITY MODELS:
A SURVEY AND AN ANALYSIS

Thesis Approved:

M. Samadzadeh
Thesis Advisor

J Chandler

[illegible]

[illegible]
Dean of the Graduate College

PREFACE

Software reliability is becoming increasingly more important in software engineering. Software reliability models have been developed to predict or estimate the reliability of software during its development. There are more than 40 software reliability models in the open literature, but no one model is best for every case under all circumstances. In this thesis, six software reliability models: Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, Shooman's model, the Littlewood-Verral model, and the Bayesian Belief Networks (BBN) model were investigated. Then the BBN model was analyzed in detail through applying it to predict the reliability of SeqWizard, a software system developed to process large-scale DNA and protein data.

For the six software reliability models investigated, the following issues were discussed for each model: a brief history, model classification, software development phase(s) applied, basic assumptions, data requirements, model form, and application scope. The six models were also compared from three aspects: factors modeled, availability of CASE tools, and advantages and disadvantages. These survey results can be used by practitioners to help make a decision when choosing among software reliability model(s).

A BBN for the reliability of SeqWizard was constructed through seven steps. The suitability of this BBN was checked by a set of hypothetical scenarios. The predicted reliability of SeqWizard was consistent with the reliability report of SeqWizard generated

by Arizona University, Purdue University, and Oklahoma State University in its beta testing stage. This indicated the suitability of the reliability BBN constructed in this thesis and the effectiveness of the BBN model for reliability prediction. In addition to reliability, the number of faults introduced to SeqWizard, the number of faults found during testing, and the number of latent faults remained in SeqWizard were also obtained from the reliability BBN. These predicted results can be used by decision-makers in software development.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor Dr. Mansur H. Samadzadeh. He provided essential guidance through my thesis work. He spent a lot of time reviewing my thesis and providing suggestions for refinement. He is my advisor at Oklahoma State University. He is also my teacher in my whole life.

I would like to thank my other committee members, Drs. G. E. Hedrick and J. P. Chandler. Their time and effort are greatly appreciated.

Great thanks also go to my parents and other family members for their understanding and encouragement. Particularly, I would like to say thank you very much to my husband, Bingdong Li, for his understanding, help, and support during my studies and my thesis work.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

Modern society is heavily dependent on automated systems to control crucial functions such as transportation, communication, utilities, and health care [Hansen et al. 99]. Software errors have caused dramatic and costly problems in recent years. Such problems include: the Year 2000 problem, the power outage at the AT&T switching facility in New York City [Pham 95], and the cost of lives (such as overdosed cancer patients from radiation therapy machines in 1986 and the death of 28 U. S. soldiers in the 1991 Gulf War [Schach 97]).

The need for reliable software increases with the increase in size and complexity of computer systems. How to get reliable software is one of the biggest questions for software engineers [Lyu 96]. There are several approaches to address this challenge. One of them is to select a proper *software reliability model* to predict or estimate the reliability of software during its development. A software reliability model specifies the general form of dependency of the failure process on the principal factors that affect it [Lyu 96]. The objectives of a software reliability model are to evaluate software quantitatively; to provide development status, test status, and schedule status; and to monitor the reliability performance in *software reliability engineering* [Neufelder 93].

A good software reliability model should have several important characteristics [Musa et al. 90]: it should give good predictions of future failure behavior and compute useful quantities, it should be simple and widely applicable, and it should be based on sound assumptions.

More than 40 software reliability models can be found in the open literature. But no one model is best for every case under all circumstances. Because it is difficult to model the interaction of the following three factors [Lyu 96]:

1) Human Factors: During either software development or software operation, humans are involved. There are different combinations of all aspects of human beings: novice or expert, hate-mode or love-mode, absent-minded or single-minded, etc. [Samadzadeh and Edwards 88] [Sommerville 97].

2) Diversity of Software Elements: Reusing existing software plays a key role in software engineering productivity, especially in the past two decades. Software reuse can be fulfilled at different levels such as object-code level, subroutine library level (executable code level), source code level, or component and package level [Zand and Samadzadeh 94]. Software reuse may introduce complexity into software system if great care is not taken. The "black box" nature of reusable software may make it hard to evaluate the reliability of the whole software system precisely.

3) Uncertain nature of software failure patterns: Software failure is known after it happens. People know that it may happen some time but do not know exactly when and how it will happen until it happens. It is hard to model the failure pattern because of this future knowledge requirement [Fenton and Neil 99].

As a result, practitioners are left with the following questions at the decision making step. Should we spend extra effort to apply a software reliability model to estimate the reliability of our software system? Which software reliability model(s) should we apply to our software system? When should we apply a software reliability model to our software system? Is there a proper automated CASE tool available for the software reliability model we need?

Typically, software reliability models are either compared based on mathematical analysis [Lyu 96] [Neufelder 93] or evaluated based on the availability of test data [Fenton and Neil 99] [Pham 95]. However, this thesis research took a different approach. This research provided a detailed survey of some current and popular software reliability models. Also, the Bayesian Belief Networks (BBN) model was analyzed through applying it to predict the reliability of SeqWizard, a software system developed by the author in the OSU Bioinformatics Lab to process large scale *DNA* and protein data.

The rest of this thesis is organized as follows. An overview of some software reliability models is given in Chapter II. A survey of six current and popular software reliability models is provided in Chapter III. The analysis of the Bayesian Belief Networks (BBN) model is given in Chapter IV. A summary and suggested future work are given in Chapter V.

CHAPTER II

OVERVIEW OF SOFTWARE RELIABILITY MODELS

More than 40 software reliability models can be found in the open literature. These models can be classified differently based on various criteria. In this thesis, software reliability models were classified into three families: 1) traditional models, 2) combination models, and 3) Bayesian models, based on the type of factors and the magnitude of each factor regarding defect prediction. The descriptions of three model families are given in the following three subsections. Traditional models are described in Section 2.1, combination models in Section 2.2, and Bayesian models in Section 2.3.

2.1 Traditional Models

The traditional software reliability models look at the impact of each fault as being of same magnitude regarding software reliability. Software reliability is changed when a fault is discovered and fixed. The main advantage of the models in this category is their simplicity. They can be easily applied if the behavior of software meets the basic assumptions. Some models have been implemented as part of CASE tools for automated applications [Lyu 96]. This family includes the following reliability models.

• *Execution Time* Model. This model uses actual execution time of software in the modeling process. Time between failures is expressed in terms of CPU time rather than elapsed calendar time. Execution time is considered more reflective of actual stress induced on software [Musa et al. 87]. Examples include Musa's Basic model and Musa's Logarithmic model [Musa et al. 90].

• *Failure Rate* Model. This model uses per-fault failure rate or *mean time to failure* (MTTF) in the modeling process. This model form assumes an exponential failure intensity function for software errors. Examples include the Jelinski-Moranda model and the Schick-Wolverton model [Dhillon 87].

• Reliability Growth Model. This model measures and predicts the improvement of reliability through debugging process. A growth function is used to represent this progress. The independent variables of the growth function can be time, number of test cases, or testing stages. The dependent variables can be reliability, failure rate, or cumulative number of errors detected [Pham 95]. Example models include the Duane growth model, the Weibull growth model [Dhillon 87], and the nonhomogeneous Poisson process (NHPP) models such as the Geol-Okumoto model and the (delayed) S-shape growth model [Liang and Trivedi 99].

• Markov Model. This model models the number of remaining faults as a stochastic counting process. When a continuous-time discrete-state Markov chain is adapted, the state of the process is the number of remaining faults, and the time between failures is the sojourning time from one state to another. If we assume that the failure rate of a program is proportional to the number of remaining faults, the linear death process and the linear birth-and-death process are two models readily available. The former assumes that the remaining errors are monotonically nonincreasing, whereas the latter allows faults to be introduced during debugging. Examples include linear death with perfect debugging (Shooman's model), linear death with imperfect debugging, and nonstationary linear death with perfect debugging [Lyu 96].

• Program Structure Model. This model views a program as a reliability network. A node represents a module or a subroutine, and the directed arc represents the program execution sequence among modules. By estimating the reliability of each node and assuming that the failure at each node is independent, we can approach the reliability of a program as a reliability network problem. Examples include the Littlewood Markov structure model and Cheung's user-oriented Markov model [Dhillon 87].

• Input Space Model. This model enumerates all sets of inputs for a computer program and determines the probability distribution of input states or the *operational profile*. Reliability is defined as the number of successful runs over the total number of runs. Examples include the Nelson model and the Shooman decomposition model [Dhillon 87]. Generally, input space models have little practical use because of their theoretical inclination [Dhillon 87].

2.2 Combination Models

The combination software reliability models are constructed from different combination of traditional models. The traditional models here are called component models of the combination model. Combination models look at the impact of each defect as being of different magnitude regarding software reliability. The difference is determined by the definition of the combination of different traditional models. The main advantage of the models in this family is their wider application range than that of the component models (traditional models). This family includes the Equally-Weighted Linear Combination Model (ELC), the Median-Oriented Linear Combination Model (MLC), the Unequally-Weighted Linear Combination Model (ULC), and the Dynamically-Weighted Linear Combination Model (DLC). These models were developed and tested by Michael Lyu and Allen Nikora with promising results [Lyu and Nikora 92].

The component models (the basic building blocks) are the Goel-Okumoto Model (GO), the Musa-Okumoto Model (MO), and the Littlewood-Verrall Model (LV). The selection of these component models is based on the facts that: 1) GO, MO, and LV are widely used and judged to perform well; 2) GO, MO, and LV represent different categories of models; and most importantly, 3) the predictive biases of GO, MO, and LV tend to be canceled when combined (GO tends to be optimistic, LV tends to be pessimistic, and MO might go either way) [Lyu and Nikora 92]. The descriptions of ELC, MLC, ULC, and DLC follow.

• Equally-Weighted Linear Combination Model (ELC). This model is formed by assuming that the three component models GO, MO, and LV have a constant equal weight. The arithmetic average of three component models' predictions is taken as the ELC model prediction. That is, $p(ELC) = 1/3\ p(GO) + 1/3\ p(MO) + 1/3\ p(LV)$. There are two major purposes for developing this model. One is to reduce the risk of relying on an individual model (traditional model) that may generate grossly inaccurate prediction. The other is to retain the simplicity of using the individual traditional models [Lyu and Nikora 92].

• Median-Oriented Linear Combination Model (MLC). In this model, the median of the predictions of all component models is the prediction of the MLC model. That is, p(MLC) = median(p(GO), p(MO), p(LV)). The justification for this model is that the choice of the median might be more moderate than the mean in some cases. It can better tolerate an erroneous prediction that is far away from the others [Lyu and Nikora 92].

• Unequally-Weighted Linear Combination Model (ULC). This model follows the idea of MLC. The prediction of ULC mainly relies on the component model that gives median performance. Unlike MLC, which ignores the results of the other two component models, ULC takes into account the prediction of the model that gives an optimistic result and the one that gives a pessimistic result. But the optimistic and pessimistic predictions will make small contributions to the final prediction. For example, p(ULC) = 1/10 O + 8/10 M + 1/10 P, where O represents an optimistic prediction, P denotes a pessimistic prediction, and M is the median prediction [Lyu and Nikora 92].

• Dynamically-Weighted Linear Combination Model (DLC). In this model, it is assumed that the applicability of any individual model may change during testing process. The weights of the component models will therefore change [Lyu and Nikora 92].

2.3 Bayesian Models

The Bayesian models take the viewpoint that software reliability should increase if no failure occurs while software is observed. This indicates the growing confidence in software by user. Reliability is the reflection of both the number of faults that have been detected and the amount of failure-free operation. Bayesian models reflect the belief that different faults have different impacts on software reliability. For instance, given a software system, when most, if not all, of the faults are clustered in a rarely executed component, we can still consider that the reliability of this software system is potentially high. The main advantage of the models in this family is their wide application scope. The Bayesian's family includes the following reliability models.

• Bayesian General Model: This model takes the viewpoint that it is more important to look at the behavior of software than to estimate the number of faults in it. It assumes that failure rate is a random process with regard to failures occurred. It also assumes a prior distribution of failure rate. This prior distribution reflects the viewpoint that one should incorporate past information in estimating reliability statistics for the present and for the future. Examples of this model type include the Littlewood-Verrall model, Kypatisi's Bayesian nonhomogeneous Poisson process model, and Liu's Bayesian geometric model [Lyu 96].

• Bayesian Belief Networks (BBN) Model: A BBN is used to model uncertainty in a problem domain in which both the quantitative and qualitative techniques are employed. It is also known as Belief Networks, Causal Probability Networks, Causal Nets, Graphical Probability Networks, Probability Cause-Effect Models, and Probabilistic Influence Diagrams. The basic idea of a BBN is that a problem is modeled as a set of nodes interconnected by a set of directed arcs to form a network. Each node in the network represents a particular occurrence or condition, called a variable. The arcs indicate the causal effect of the variables on each other. Each node contains a set of states of the random variable it represents and a *conditional probability table* (*cpt*). The cpt of a node contains probabilities of the node being in a specific state, given the states of its parents. The probabilities in the CPT indicate the strength of link between nodes. These probabilities are used to calculate the probability distribution of all nodes in a BBN when new evidence is entered. Actually a BBN is a directed acyclic graph (DAG). This structure prevents circular logic because the arcs represent causal relationships between nodes. In practice, this structure prevents the algorithm from getting into an infinite loop or deadlock. When a BBN is established, all the arcs are seeded with the initial estimation of probabilities. When new evidence is provided, the network automatically updates the probabilities of all arcs. Recalculation of all the probabilities continues to propagate across the network, fine-tuning the accuracy of all probabilities. This ability to revise the probability of an event, based on the new evidence and the old status, is the main advantage of the BBN [Ziv and Richardson 97] [Yu et al. 99].

CHAPTER III

A SURVEY OF SOME SOFTWARE RELIABILITY MODELS

The purpose of this survey is to provide information for practitioners to choose proper software reliability models. Six current and popular software reliability models are investigated in Section 3.1. Comparisons of six models are given in Section 3.2.

3.1 Investigation of Six Software Reliability Models

In this section, six current and popular software reliability models were investigated. These models included Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, Shooman's model (these four models belong to the traditional model family), the Littlewood-Verral model, and the Bayesian Belief Networks (BBN) model (these two models belong to the Bayesian model family). Since combination models were still under development, no model was selected from this model family for detailed investigation.

The selection of the above models was based on their applicability and their representative effects. Musa's models "have had the widest distribution among the software reliability models" [Lyu 96]. The Jelinski-Moranda model "has the most articles written on it in the literatures on software reliability because of its sound theoretical foundation" [Lyu 96]. The Geol-Okumoto NHPP model "has strong influence on software reliability modeling" [Liang and Trivedi 99]. It is one of the practical models that model real situations thoroughly and provide meaningful results [Lyu 96]. Shooman's model represents the Markov model type, which is considered as "a general way to model software failure process" [Pham 95]. The Littlewood-Verral model is the software reliability model recommended by the American Institute of Aeronautics and Astronautics (AIAA) as recommended practice standard (AIAA 93) [Lyu 96]. The Bayesian Belief Networks (BBN) model has attracted much recent attention in diagnostic and predictive analysis and it has been considered a powerful tool for reasoning under uncertainty [Fenton and Neil 99].

For each selected software reliability model, the following issues were discussed: 1) a brief history, 2) model classification, 3) software development phase(s) applied, 4) basic assumptions, 5) data requirements, 6) model form, and 7) application scope. Also, some comments on the successes/failures of each model were included. The rest of this section was organized according to these issues.

Following two formulas were used in each model.

Formula 1: $R(t) = \exp [- \int_0^t \lambda(z)\, dz]$

Formula 2: $MTTF = \int_0^\infty R(t)\, dt$

where R(t) is software reliability at time t, λ(t) is *failure intensity function* or *hazard rate*, and MTTF is *mean time to failure*. For each model described below, the failure intensity function, λ(t), was given. The R(t) and MTTF for each model can be obtained by Formula 1 and Formula 2, respectively.

3.1.1 Musa's Models

3.1.1.1 Brief History

Musa's models were created and developed by John Musa of AT&T Bell Laboratories with his colleagues in 1970's. There are two types of Musa's model. One is Musa's Basic model and the other is Musa's Logarithmic model [Musa et al. 90].

3.1.1.2 Model Classification

Both Musa's Basic and Logarithmic models are execution time models in the traditional model family.

3.1.1.3 Software Development Phase(s) Applied

Musa's models can be applied in unit test phase in the software life cycle.

3.1.1.4 Basic Assumptions

Musa's models model software reliability under the following basic assumptions.

(1) Cumulative number of failures by time t follows a Poisson process.

(2) There are a finite number of total errors estimated in software. This number is not necessarily fixed.

(3) Errors are rectified before software test continues.

(4) Hazard rate for a single fault is constant.

(5) Failure rate is proportional to residual software errors.

3.1.1.5 Data Requirements

Musa's models require actual execution time that software fails or elapsed time between failures.

3.1.1.6 Model Form

The failure intensity function, $\lambda(t)$, for Musa's Basic model is given as follows.

$$\lambda(t) = \lambda_0 \exp(-\beta_0 \times t)$$

where t is execution time, λ_0 is initial failure intensity at the start of execution, $\beta_0 = \lambda_0/\nu_0$, and ν_0 is total number of failures that would occur in infinite time.

The failure intensity function, $\lambda(t)$, for Musa's Logarithmic model is given as follows.

$$\lambda(t) = \lambda_0 / (\lambda_0 \theta t + 1)$$

where t is execution time, λ_0 is initial failure intensity at the start of execution, and θ is failure intensity decay parameter.

3.1.1.7 Application Scope

Musa suggested that the choice of the two models in any given application depends on several factors as summarized in Table I.

Table I. Factors for the Choice Between Musa's Basic and Logarithmic Models [Musa et al. 90]

Purpose of application	Basic Model	Logarithmic Model
Studies or predictions before execution	X	
Studying effects of software engineering technology (through study of faults)	X	
Program size changing continually and substantially	X	
Highly nonuniform operational profile		X
Early predictive validity important		X

3.1.1.8 Comments

Musa's models have been applied in many diverse fields. They have several advantages: their predicting results are generally satisfactory, they are simple and easy to understand, they are thoroughly developed, their parameters have a clear physical interpretation, and they can handle dynamic systems [Musa et al. 90].

3.1.2 The Jelinski-Moranda Model

3.1.2.1 Brief History

The Jelinski-Moranda model was created and developed by Paul L. Jelinski and Z. Moranda in 1972 [Dhillon 87].

3.1.2.2 Model Classification

The Jelinski-Moranda model is a failure rate model in the traditional model family.

3.1.2.3 Software Development Phase(s) Applied

The Jelinski-Moranda model can be applied in test phase in the software life cycle.

3.1.2.4 Basic Assumptions

The Jelinski-Moranda model models software reliability under the following basic assumptions.

(1) There are a constant number of lines of code.

(2) The operational profile of software is consistent.

(3) Every fault has the same chance to be encountered during software operation.

(4) Fault detection rate remains constant over intervals between fault occurrences.

(5) Fault detection rate is proportional to current fault content of software.

(6) Each detected error is corrected without delay.

(7) Failures are independent.

3.1.2.5 Data Requirements

Data required for using the Jelinski-Moranda model are elapsed time between failures or the actual times that software fails.

3.1.2.6 Model Form

The failure intensity function, $\lambda(t)$, is given by the following formula.

$$\lambda(t_j) = C\,(E_i - (j - 1))$$

where C is a constant of proportionality, E_i is number of initial software errors, and t_j is time between detection of the jth and (j - 1)th software errors. With the aid of maximum likelihood estimation method, the values of parameters E_i and C can be estimated as follows.

$$\sum_{j=1}^{k} 1/(E_i - (j-1)) = k \sum_{j=1}^{k} t_j \Big/ \sum_{j=1}^{k} (E_i - (j-1))\, t_j$$

$$C = k \Big/ \sum_{j=1}^{k} (E_i - (j-1))\, t_j$$

where k is number of intervals to detect software errors and t_1, t_2, ..., t_j are sampling intervals of time between successive software errors [Dhillon 87].

3.1.2.7 Application Scope

Nowadays the Jelinski-Moranda model is not being used directly. But variants developed from it are widely used [Dhillon 87].

3.1.2.8 Comments

The Jelinski-Moranda model is one of the earliest models developed. The importance of this model is mainly in setting a framework in this modeling area [Lyu 96].

3.1.3 The Geol-Okumoto NHPP Model

3.1.3.1 Brief History

The Geol-Okumoto NHPP model was proposed and developed by Amrit Goel and Kazu Okumoto in 1979 [Liang and Trivedi 99]. The (delayed) S-shaped NHPP is a variant of this model developed by Yamada in 1982 [Lyu 96].

3.1.3.2 Model Classification

The Geol-Okumoto NHPP model is a reliability growth model in the traditional model family.

3.1.3.3 Software Development Phase(s) Applied

The Geol-Okumoto NHPP model can be applied in pre-release phases in the software life cycle.

3.1.3.4 Basic Assumptions

The Geol-Okumoto NHPP model models software reliability under the following basic assumptions.

(1) Cumulative number of failures by time t follows a Poisson process.

(2) Number of faults detected in each time interval is independent for any finite collection of time intervals.

(3) Defects are repaired immediately when they are discovered.

(4) Defect repair is perfect. That is, no new defect is introduced during test.

(5) No new code is added to software during test.

(6) Each unit of execution time during test is equally likely to find a defect if the same code is executed at the same time.

3.1.3.5 Data Requirements

The Geol-Okumoto NHPP model requires number of faults counted in each test time interval and the completion time of each period the software is under observation.

3.1.3.6 Model Form

The mean value function $\mu(t)$ is given by following formula.

$$\mu(t) = a\,(1 - e^{-bt})$$

where t is failure occurrence time, a is expected total number of faults to be eventually detected, and b is a positive constant.

The failure intensity function, $\lambda(t)$, which is the derivative of $\mu(t)$, is obtained as follows.

$$\lambda(t) = a\,b\,e^{-bt}$$

where a is expected total number of faults to be eventually detected, b is a positive constant, and t is failure occurrence time.

3.1.3.7 Application Scope

The Geol-Okumoto NHPP model can be used to determine an optimal release time for software.

3.1.3.8 Comments

Goel and Okumoto also adapted this model to use time of fault occurrences instead of fault counts. Optimal release time was determined based on the cost of finding and fixing a fault in the testing environment versus the operational environment. The (delayed) S-shaped NHPP variant has been successfully applied in several applications [Liang and Trivedi 99].

3.1.4 Shooman's Model

3.1.4.1 Brief History

Shooman's model was created and developed by Martin Shooman of the New York Polytechnic Institute in 1972 [Neufelder 93].

3.1.4.2 Model Classification

Shooman's model is a Markov model in the traditional model family.

3.1.4.3 Software Development Phase(s) Applied

Shooman's model can be used in system integration test phase in the software life cycle.

3.1.4.4 Basic Assumptions

Shooman's model models software reliability under the following basic assumptions.

(1) Total number of machine instructions remains constant.

(2) Total number of software errors decreases directly as errors are rectified.

(3) No new errors are introduced during the debugging process.

(4) Residual errors are given by subtracting cumulative errors rectified from total number of errors initially presented.

(5) Hazard function is proportional to remaining number of errors.

3.1.4.5 Data Requirements

Data required for using Shooman's model are total number of machine instructions in software, execution time of software in a given time period, and total number of errors during a given time period.

3.1.4.6 Model Form

The failure intensity function, $\lambda(t)$, is defined by the following formula.

$$\lambda(t) = \alpha\, E_t(y)$$

where t is execution time, α is a proportionality constant, y is debugging time since the beginning of system integration, and $E_t(y)$ is total number of errors remaining in software at time y. $E_t(y)$ is given as follows.

$$E_t(y) = E_{ti} / I_m - E_{cu}(y)$$

where E_{ti} is total number of initial errors at time $y = 0$, which can be approximated using MTTF, I_m, and $E_{cu}(y)$. I_m is total number of machine language instructions in software. $E_{cu}(y)$ is cumulative number of software errors in the time interval $[0,y]$. α is given by following formula.

$$\alpha = \theta / (t\, E_t(y))$$

where θ is total number of errors during debugging period of $[0,y]$, t is execution time of software corresponding to debugging period of $[0,y]$, and $E_t(y)$ is given above.

3.1.4.7 Application Scope

Shooman's model can be used for any software development as long as the process meets the Markov chain.

3.1.4.8 Comments

It has been shown that Shooman's model is more effective if used later in the software life cycle. The main reason for this is that earlier in development, particularly during debugging and unit test, error rate will probably be increasing over time, in which case Shooman's model cannot be applied [Neufelder 93].

3.1.5 The Littlewood-Verrall Model

3.1.5.1 Brief History

The Littlewood-Verrall model was created and developed by B. Littlewood and J. L. Verrall in the 1970's [Lyu 96].

3.1.5.2 Model Classification

The Littlewood-Verrall model is a Bayesian general model in the Bayesian model family.

3.1.5.3 Software Development Phase(s) Applied

The Littlewood-Verrall model can be applied in test and release phases in the software life cycle.

3.1.5.4 Basic Assumptions

The Littlewood-Verrall model models software reliability under the following basic assumptions.

(1) Successive execution times between failures are independent with parameter ξ that is related to historical data.

(2) Function $\psi(i)$ is an increasing function of programmers' quality and task's difficulty. A good programmer would have a more rapidly increasing function than a poor one.

(3) Software is operated in the same manner as expected. That is, software is not frequently used under extreme situations.

3.1.5.5 Data Requirements

The Littlewood-Verrall model requires time between failure occurrences.

3.1.5.6 Model Form

The model form of the Littlewood-Verrall model varies based on the definition of ξ, a parameter relating to historical data, and $\psi(i)$, a function indicating programmers' quality and system's complexity. There are two basic forms: linear form and quadratic form.

In the linear form, the failure intensity function, $\lambda(t)$, is given as follows.

$$\lambda(t) = (\alpha - 1) / (\beta_0^2 + 2 \beta_1 t (\alpha - 1))^{1/2}$$

where t is failure occurrence time, α, β_0, and β_1 are positive constants related to ξ and $\psi(i)$.

In the quadratic form, the failure intensity function, $\lambda(t)$, is given as follows.

$$\lambda(t) = (\nu_1 / (t^2 + \nu_2)^{1/2}) \times ((t + (t^2 + \nu_2)^{1/2})^{1/3} - (t - (t^2 + \nu_2)^{1/2})^{1/3})$$

where t is failure occurrence time, $\nu_1 = (\alpha - 1)^{1/3} / (18 \beta_1)^{1/3}$ and $\nu_2 = 4 \beta_0^3 / (9 (\alpha - 1)^2 \beta_1)$. Here α, β_0, and β_1 are positive constants related to ξ and $\psi(i)$.

3.1.5.7 Application Scope

The Littlewood-Verrall model is now widely applied in software reliability estimation and prediction. Successful applications of this model alone, or combinations of this model with other models, show that the Littlewood-Verrall model is effective to estimate the reliability of software after its completion and before its release [Fenton and Neil 99].

3.1.5.8 Comments

The main characteristic of the Littlewood-Verrall model series is that fault correction process allows the probability that software could be less reliable than before. That is, the new version of software may be either better or worse than its predecessor. Due to this uncertainty, the model form of the Littlewood-Verrall model becomes very complicated but flexible [Lyu 96].

3.1.6 The Bayesian Belief Networks (BBN) Model

3.1.6.1 Brief History

Bayes' theorem, the underlying theory of BBN, was developed by Thomas Bayes in the 18th century [Lauritzen and Spiegelhalter 88]. It was seldom used in computer science before the 1980's. The success of applying recent algorithms and software tools made it possible to build and execute realistic models. Bayesian Belief Networks (BBN) model has attracted much recent attention as a possible solution to the problems of decision support under uncertainty [Cowell et al. 99] [Fenton and Neil 99].

3.1.6.2 Model Classification

The Bayesian Belief Networks (BBN) model is in the Bayesian model family.

3.1.6.3 Software Development Phase(s) Applied

The BBN model can be applied in all phases in the software life cycle. Defect prediction can be explained in two stages [Fenton and Neil 99]. The first stage covers the phases of requirement analysis, specification development, design, and implementation. The second stage covers the phases of test and pre-release.

3.1.6.4 Basic Assumptions

The Bayesian Belief Networks (BBN) model models software reliability under the following assumptions.

(1) Software defects are not directly caused by program complexity alone.

(2) Causal factors for the presence of defects in software include: a) difficulty of problem; b) complexity of designed solution; c) programmer/analyst skills; and d) design methods and procedure applied.

3.1.6.5 Data Requirements

All data in the software life cycle can be useful for this model.

3.1.6.6 Model Form

A BBN is defined as a triple (N, E, P), where N is a set of nodes, $E \subseteq N \times N$ a set of edges, and P a set of probabilities [Cowell et al. 99]. Each node in N is labeled by a random variable v_i, where $1 \leq i \leq |N|$. Each node v_i has a set of states associated with it. The states of a node can be Boolean values, discrete labels, discrete numbers, intervals, or continuous values. Each directed edge $e_i = < s_i, t_i > \in E$ indicates causal influence from source node s_i (parent node) to target node t_i (child node). For each node t_i, the strengths of causal influences from its parent s_i are quantified by a conditional probability distribution $p(t_i | s_i)$, specified in an $m \times n$ matrix where m is the number of states of t_i, and n is the number of states of s_i. This $m \times n$ matrix is called the *conditional probability table (CPT)* of node t_i. Each $p_i \in P$ is the conditional probability of a node being in a specific state given the states of its parents. If a node has no parents, its probability is unconditional. Its CPT is actually a uniform probability distribution if not specified otherwise. For each node in a BBN, the sum of probabilities of all states should be 1. This indicates that the BBN model is subject to the standard axiom of probability theory.

3.1.6.7 Application Scope

A BBN is generally applied to problems when there is uncertainty in the data or in the knowledge about the domain. It has been applied to problems that require diagnosis of problems from a variety of input data. BBN is generally used in: medical diagnostic systems such as MUNIN (a medical reasoning system); analysis in the natural, biological, and social sciences; real-time weapons scheduling; computer processor (Intel) fault diagnosis; complex machinery monitoring system such as jet engines, electronic power generator, and copy machines; trouble-shooting mechanisms such as the help wizard in Microsoft Office [Hansen et al. 99]. BBN is also used in expert systems [Cowell et al. 99]. In software engineering, BBNs have been applied to estimate uncertainties in software testing and maintenance [Ziv and Richardson 97], to analyze the dependability of safety-critical software systems [Bouissou et al. 99], to predict software quality [Neil and Fenton 96], and to support decision-making during software development [Fenton and Neil 99]. BBN seems to have potential to combine different approaches of defect prediction into a single model.

3.1.6.8 Comments

A BBN represents a complete probabilistic model of the system. This is due to the fact that the joint probability distribution of any elementary system can be derived using the local conditional probability distributions and network topology [Cowell et al. 99]. Figure 1 is an example of the joint of subsystems. In Figure 1, system 1 and system 2 are joined together through the common node "#faults". After the joint, the probability of each node in Figure 1 (c) will be updated accordingly. The updating will be propagated over the network to ensure the accuracy of the probability of each node in the whole network [Cowell et al. 99]. This property suggests the use of BBN models for evaluating different components in complicated software systems.

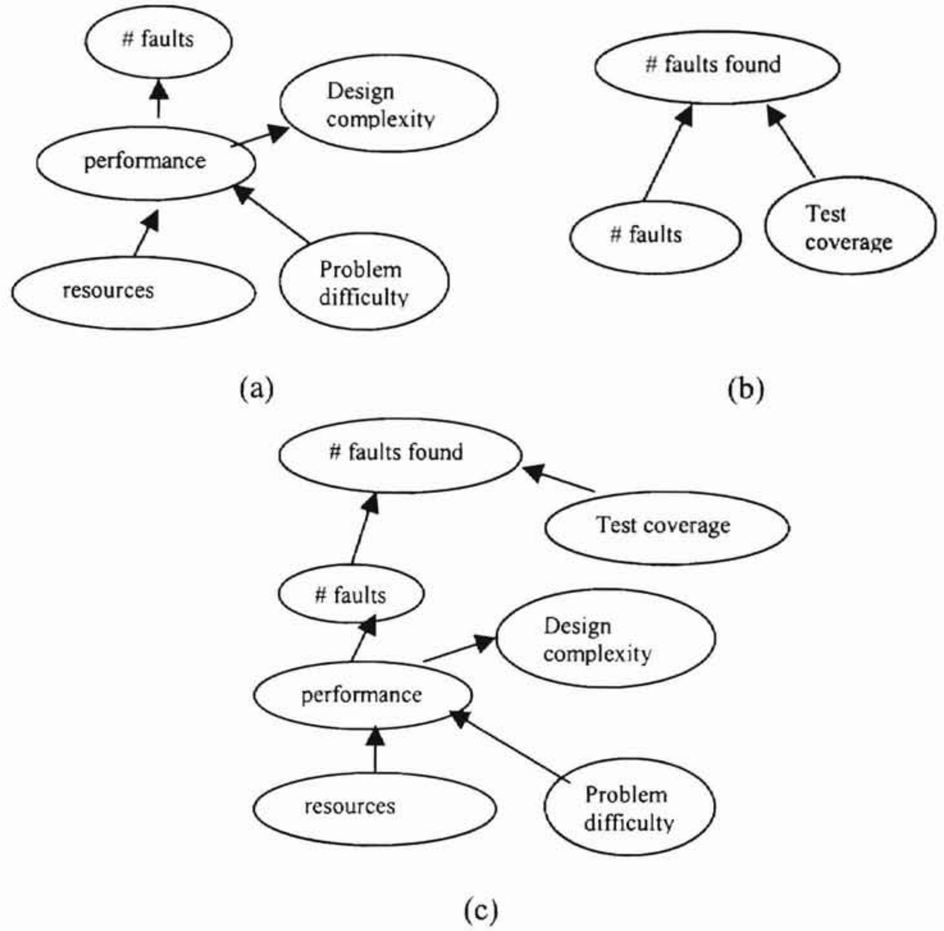


Figure 1. The Joint of Subsystems: (a) system 1, (b) system 2, and (c) the joint of system 1 and system 2

The advantages of using BBN are summarized below.

(1) Specifying complex relationships using conditional probability statements.

(2) Using 'what–if?' analysis and forecasting the effects of process changes.

(3) Easy understanding of seemingly contradictory reasoning via graphical format.

(4) Explicit modeling 'ignorance' and uncertainty in estimates.

(5) Using subjectively or objectively derived probability distributions.

(6) Forecasting with missing data.

3.2 Comparisons of Six Software Reliability Models

In this section, Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, Shooman's model, the Littlewood-Verral model, and the Bayesian Belief Networks (BBN) model are compared from three aspects. The comparison from the aspect of factors modeled is given in Section 3.2.1, the comparison from the aspect of availability of CASE tools is given in Section 3.2.2, and the comparison from the aspect of advantages and disadvantages is given in Section 3.2.3.

3.2.1 Comparison of Factors Modeled

Software reliability models specify the general form of dependency of the failure process on the principal factors that affect it in order to predict or estimate software reliability [Lyu 96]. Such factors are involved in fault introduction, fault removal, and operational environment. In the software life cycle, fault introduction may happen in requirement analysis phase, specification development phase, design phase, or implementation phase. In addition, fault introduction may happen due to problem complexity. Fault removal happens in test phase. Operational environment refers to proper/improper usage of software or the consistency of software operational profile.

Since no standard is available, different software reliability models model various factors [Fenton et al. 99]. Factors modeled by a software reliability model determine its reliability measurement type: reliability prediction, reliability estimation, or reliability prediction and estimation. The measurement type is very important to practitioners. Factors modeled by six software reliability models are compared in Table II.

Table II. Factors Modeled by Six Software Reliability Models
(JM – Jelinski-Moranda, GO – Geol-Okumoto, LV – Littlewood-Verral)

	Problem complexity	Software developers	Software solution	Software testers	Operational profile
Musa's Models			Execution time, number of failures	Competence of testers	
The JM Model			LOC, failure rate	Competence of testers	Consistency
The GO NHPP Model			Number of faults, times software fail	Competence of testers	
Shoo-man's Model			Number of machine instructions, execution time, total errors	Competence of testers	
The LV Model	Difficulty of task	Programmers' competence	Time between failures	Effort of testers	Consistency
The BBN Model	Complexity of problem	Experience and effort of analysts, specifiers, designers, and programmers	Estimation of latent faults	Experience and effort of testers	Consistency

Table II showed that Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, and Shooman's model model two kinds of factors: 1) software failure data obtained in test and operation phases, and 2) the competence of software testers. From software reliability measurement point of view, these models mainly focus on software reliability estimation. The Littlewood-Verral model and the BBN model model factors in each phase of software life cycle when failure data may or may not available. From software reliability measurement point of view, the Littlewood-Verral model and the BBN model focus on both software reliability prediction and estimation.

3.2.2 Comparison of Availability of CASE Tools

After software reliability model(s) are chosen for specific purposes, one of the many decisions is the choice of a CASE tool. In this section, the following software reliability CASE tools were investigated: AT&T Software Reliability Toolkit (AT&T SR Toolkit) [Lyu 96], Statistical Modeling and Estimation of Reliability Function for Software (SMERFS) [Musa et al. 90], Software Reliability Program (SoRel) [Lyu 96], Computer-Aided Software Reliability Estimation (CASRE) [Lyu 96], Statistical Reliability Modeling Program (SRMP) [Neufelder 93], Economic Stop Testing Model (ESTM) [Musa et al. 90], the GOEL tool [Neufelder 93], the HUGIN tool [Hugin 90], and the Norsys tool [Norsys 95]. The availability of CASE tools for six models is compared in Table III.

Table III. Availability of CASE Tools for the Six Models
(JM – Jelinski-Moranda, GO – Geol-Okumoto, LV – Littlewood-Verral)

	Musa's Models	The JM Model	The GO NHPP Model	Shooman's Model	The LV Model	The BBN Model
AT&T SR Toolkit	X					
SMERFS	X	X	X		X	
SoRel			X		X	
CASRE	X	X	X	X	X	
SRMP	X	X	X	X	X	
ESTM			X			
GEOL			X			
HUGIN						X
Norsys						X

For each CASE tool in Table III, the supplier of the tool, original release time, hardware requirement, operating systems requirement, memory size requirement, user interface, and current price are provided in Table IV (abbreviations in this table are given in the Glossary).

Table IV. Comparison of Software Reliability CASE Tools

	Supplier	Original release	Hard-ware	Operating system	Size	User interface	Current price
AT&T SR Toolkit	AT&T Bell Lab.	1991	Any platform	UNIX, MS/DOS	120 K	Command -driven	$60
SME-RFS	NSWC	1983	DEC VAX, IBM PC	DEC VMS, MS-DOS 3.0	256 K	Menu-driven	$59.95
SoRel	LAAS	1991	Macintosh II	Macintosh	200 K	Command or Menu-driven	Free
CASRE	NASA	1995	IBM PC	MS-DOS 5.0	8MB	Menu-driven	$136
SRMP	RSC Ltd.	1988	IBM PC	MS-DOS 3.0	500 K	Command -driven	$5000
ESTM	Bellcore	1987	Sun, HP, Dec Worksta -tion	UNIX	Not Available	Integrated system	Call Bellcore
GEOL	DACS	1987	IBM PC	MS-DOS 2.11	256	Menu-driven	$50
HUGIN	HUGIN Expert	1990	IBM PC	Windows 95 or higher	10 MB	GUI	$3000-$25000
Norsys	Norsys	1995	IBM PC	Windows 95 or higher	12 MB	GUI	$1000-$2500

3.2.3 Comparison of Advantages and Disadvantages

The advantages and disadvantages of six models are compared in Table V.

Table V. Advantages and Disadvantages of the Six Models
(JM – Jelinski-Moranda, GO – Geol-Okumoto, LV – Littlewood-Verral)

	Advantages	Disadvantages
Musa's Models	Simple and easy to understand; clear physical interpretation of parameters; low cost to implement; several CASE tools are available	Limited application scope; some assumptions are too simple; some causal factors are not modeled; cannot handle uncertainty
The JM Model	Simple and easy to understand; sound theoretical foundation; low cost to implement; several CASE tools are available	Limited application scope; some assumptions are too simple; some causal factors are not modeled; cannot handle uncertainty
The GO NHPP Model	Simple and easy to understand; low cost to implement; many CASE tools are available	Limited application scope; some assumptions are too simple; some causal factors are not modeled; cannot handle uncertainty
Shoo-man's Model	Simple and easy to understand; low cost to implement	Limited application scope; some assumptions are too simple; some causal factors are not modeled; cannot handle uncertainty; few CASE tools are available
The LV Model	Wide application scope; models most causal factors; practical assumptions; considers uncertainty; several CASE tools are available	Complex model form; dose not give a clear solution to handle uncertainty; difficult to implement
The BBN Model	Wide application scope; models most causal factors; practical assumptions; can handle uncertainty	Complex model form; difficult to implement; few CASE tools are available

CHAPTER IV

AN ANALYSIS OF THE BAYESIAN BELIEF NETWORKS MODEL BY APPLYING IT TO SeqWizard

A Bayesian Belief Network (BBN) enables users to model and reason about uncertainty. It has the advantage of combining an intuitive visual representation (a directed acyclic graph or DAG) with a sound mathematical basis (probability theory such as conditional probability and Bayes' theorem) [Pearl 97].

Conditional probability is defined as follows.

$$P(A \mid B) = P(A, B) / P(B)$$

where $P(A \mid B)$ is the probability that event A happens under the condition that event B is known, $P(A, B)$ is the probability that events A and B happen at the same time, and $P(B)$ is the probability that event B happens.

Bayes' theorem is given as follows.

$$P(A \mid B) = P(B \mid A) \times P(A) / P(B)$$

where $P(A \mid B)$ is called posterior belief of event A given event B, $P(A)$ and $P(B)$ are called prior belief of event A and probability of event B, respectively, and $P(B \mid A)$ is called the likelihood that event B will occur if event A is true.

The importance of Bayes' theorem is that it connects two different probabilities $P(A \mid B)$ and $P(B \mid A)$. In $P(A \mid B)$, B is evidence and A is uncertain. In $P(B \mid A)$, A is evidence and B is uncertain. Bayes' theorem propagates the effects of evidence through a network of variables in all directions [Lauritzen and Spiegelhalter 88]. This propagation ability with other features makes BBN a powerful tool in diagnostic and predictive analysis [Cowell et al. 99].

In this chapter, the BBN software reliability model was used to predict the reliability of the SeqWizard software. The SERENE 1.0 Demo tool [SERENE 90] was used for BBN construction and probability computation.

This chapter is organized as follows. A brief introduction to SeqWizard is provided in Section 4.1. The analysis of the reliability BBN model for SeqWizard is given in Section 4.2. A discussion of this analysis is given in Section 4.3.

4.1 Introduction to SeqWizard

SeqWizard is a software system developed by the author in the OSU Bioinformatics Lab to automatically process large-scale DNA and protein data.

SeqWizard software contains three major components: SeqProcessor, SeqAnalyzer, and SeqDatabase. SeqProcessor processes DNA or protein sequence data. SeqAnalyzer analyzes the similarity of two or more DNA or protein sequences. SeqDatabase generates relational databases for users. Users can communicate with the SeqWizard software system from web based graphical user interfaces.

SeqWizard provides following key functionality: 1) searching and browsing from within the working group or from the outside world; 2) processing large-scale DNA or protein sequence data and generating relational databases; 3) functionally classifying

DNA or protein sequences based on their similarities with the DNA or protein sequences that have known functions.

4.2 The Analysis of the Reliability BBN Model for SeqWizard

This analysis included seven steps: 1) identifying the key entities in SeqWizard, 2) determining the key attributes of the entities relating to reliability, 3) grouping together related attributes, 4) determining appropriate *idioms* for each group of attributes, 5) defining the conditional probability table for each node in each idiom, 6) building the complete reliability BBN for SeqWizard, and 7) predicting the reliability of SeqWizard. The rest of this section is organized according to these steps. In Section 4.2.1, identification of the key entities in SeqWizard is provided. In Section 4.2.2, determination of the key attributes relating to reliability is given. In Section 4.2.3, related attributes are grouped together in order to choose a proper idiom. In Section 4.2.4, an appropriate idiom is determined for each group of attributes. In Section 4.2.5, the conditional probability table for each node in each idiom is defined. In Section 4.2.6, all the idioms are joined together to build a complete reliability BBN for SeqWizard. Finally in Section 4.2.7, the reliability of SeqWizard is predicted.

4.2.1 Identifying Key Entities in SeqWizard

Entities are things or events of interest. In SeqWizard, there are three types of entities: resource entities, process entities, and product entities. Resource entities include software designers, programmers, and testers. Process entities include the problem identification, requirement analysis, specification development, design process, coding process, and testing process of SeqWizard. Here we mainly focus on design process, coding process, and testing process. Product entity is the SeqWizard software. The SeqWizard software consists of the design generated, the code produced, the programmer's guide, the user's manual, and the test report.

The relationships among entities of SeqWizard are shown in Figure 2.

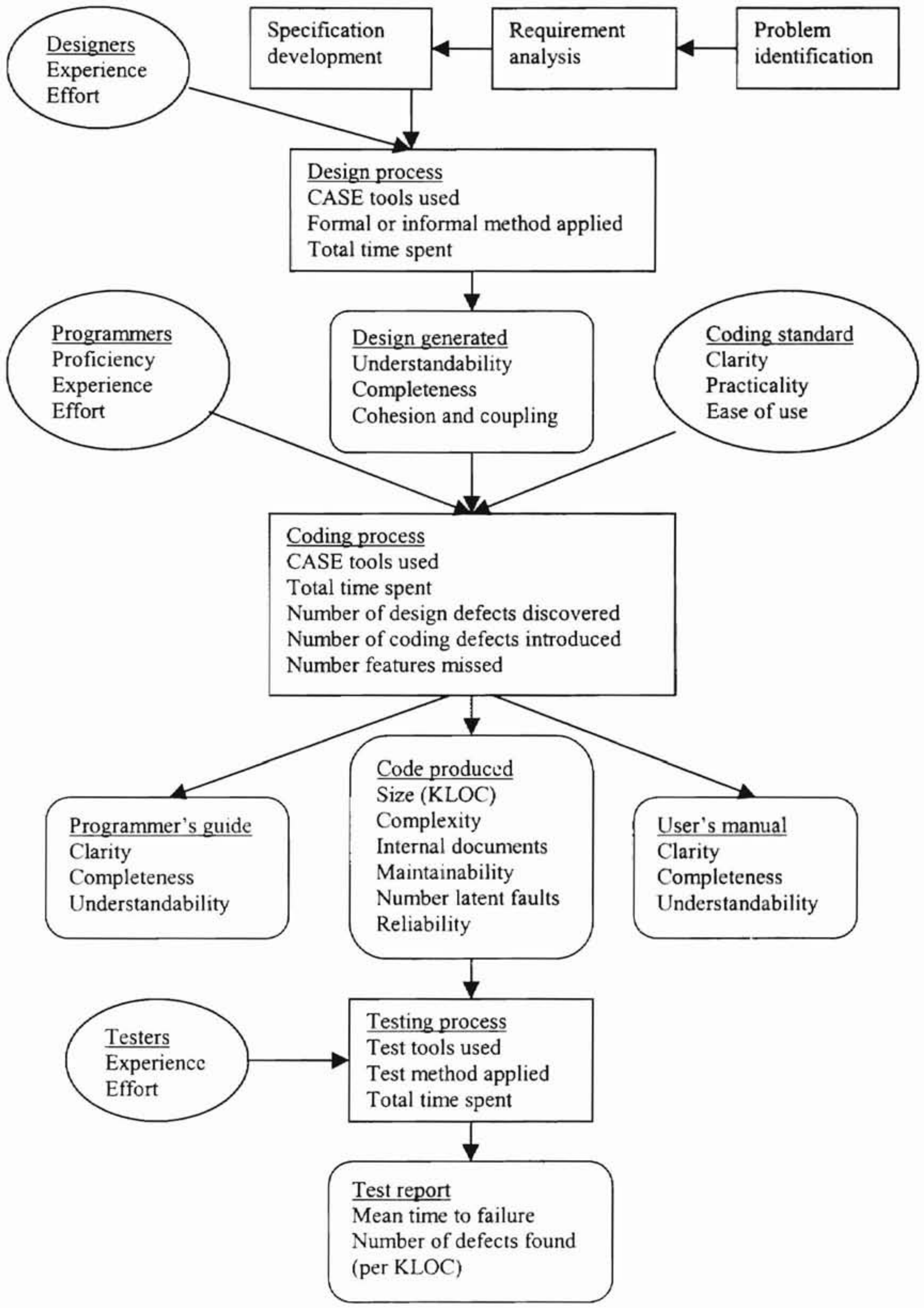


Figure 2. Relationships Among Entities(⬭: resource; ▭ : process; ▢ : product)

4.2.2 Determining Key Attributes Relating to Reliability

Attributes are properties of entities. In SeqWizard, the key entities were identified in Section 4.2.1. In this section, the key attributes of each entity relating to reliability are determined and listed in Table VI.

Table VI. Key Attributes in SeqWizard Relating to Reliability

Entity type	Entity	Key attributes relating to reliability	Description of key attributes in SeqWizard
Resource	Designers	Experience	About five years of design experience
		Effort	30% of the total development time*
	Programmers	Experience	1-2 years of programming experience
		Effort	50% of the total development time
	Testers	Experience	1-3 years of testing experience
		Effort	20% of the total development time
Process	Design	Design quality	
	Coding	Coding accuracy	
	Testing	Testing accuracy	
Product	SeqWizard	Reliability	Attribute to be predicted
		Number of latent faults per KLOC**	60 crucial faults were found in SeqWizard during testing
		Operational profile	Changed clearly but not dramatically
		Problem complexity	25 KLOC** in SeqWizard

*Total development time – time includes design, coding, and testing

**KLOC – thousand lines of code

For resource entities in Table VI, Designers' Experience is the ability to select design methods, CASE tools, and design strategies. Design Effort is the time spent on design. Designers' Experience and Effort directly influence design quality. Programmers' Experience is the ability to program defensively and the ability to improve code testability. Programmers' Effort is the time spent on coding. Programmers' Experience and Effort directly influence coding accuracy. Testers' Experience is the ability to make test plan, to select test strategies, and to choose test cases. Testers' Effort is the time spent on testing. Testers' Experience and Effort directly influence testing accuracy.

For process entities in Table VI, Design quality is the completeness and accuracy of mapping a real world problem to a software design. Software faults are actually design faults [Lyu 96]. Design faults are usually hard to visualize, detect, and correct. Coding accuracy is the correctness of mapping a software design to a software solution. Testing accuracy indicates number of crucial faults found in software. Design quality, coding accuracy, and testing accuracy are directly related to number of latent faults in software.

For product entity in Table VI, Reliability is the attribute to be predicted for SeqWizard. Reliability is defined as the probability of failure-free operations. Number of latent faults per KLOC is the direct source of software failure. Operational profile is a set of operations and the frequency of each operation specified for software. During software operation, if the operational profile is changed, software reliability will change. Problem complexity is an attribute that influence design quality and testing accuracy. The more complex the problem is, the harder it is for software designers to catch all its features. The more complex the problem is, the harder it is for testers to find defects in software.

4.2.3 Grouping Together Related Attributes

The key attributes listed in Table VI are grouped together in Table VII with rationales.

Table VII. Grouping Related Attributes

Group	Attributes in group	Rationale for grouping
1	Reliability #latent faults per KLOC Operational profile	The fewer the number of latent faults are and the more consistent the operational profile is, the higher will the reliability be.
2	#latent faults per KLOC #faults introduced per KLOC #faults found per KLOC	The #latent faults per KLOC equals #faults introduced per KLOC minus #faults found and fixed per KLOC.
3	#faults found per KLOC #faults introduced per KLOC Testing accuracy	The more #faults introduced in software and the higher the testing accuracy, the more likely it is to find faults in software.
4	#faults introduced per KLOC Design quality Coding accuracy	The higher the design quality, the fewer design faults are introduced. The higher the coding accuracy, the fewer coding bugs.
5	Design quality Design experience Design effort Problem complexity	The more experience designers have and the more effort spent, the higher the design quality. The more complex the problem, the harder for designers to capture all its features.
6	Coding accuracy Coding experience Coding effort	The more experience programmers have and the more effort spent on coding, the fewer bugs are introduced into the code.
7	Testing accuracy Testing experience Testing effort Problem complexity	The more experience testers have and the more effort spent, the more likely to find crucial bugs. The more complex the problem is, the harder it is to find defects in software.

In Table VII, related attributes are grouped together in order to determine an appropriate idiom for each group in the next section. Each attribute listed in Table VII would correspond to a node in the BBN. Some attributes are grouped into several groups because they play various roles in different group. Another reason was that different idioms can be joined together by these common attributes.

4.2.4 Determining Idioms

Different attributes in SeqWizard were grouped together in Section 4.2.3. In this section, an appropriate idiom is determined for each group of attributes. An introduction to idioms and how to choose appropriate idioms is given in Section 4.2.4.1. Determination of an appropriate idiom for each group of attributes is given in Section 4.2.4.2.

4.2.4.1. What are Idioms and How to Choose Appropriate Idioms?

Idioms are generally applicable building blocks for constructing BBNs [Neil et al. 99]. Idioms specify the relationships among nodes. The main purpose of using idioms is that idioms act as a library of patterns for BBN development. Using idioms emphasizes reuse. Five categories of idioms had been identified based on generic uncertain reasoning [Neil et al. 99]. These five categories of idioms are described below.

- The Definition/Synthesis Idiom. This idiom models the synthesis or combination of many nodes into one node for the purpose of organizing the BBN. It also models the deterministic or uncertain definition between variables. This idiom emphasizes definitional reasoning such as "what something is".

- The Cause-Consequence Idiom. This idiom models the uncertainty of an uncertain causal process with observable consequences. This idiom emphasizes production and transformation of something.
- The Measurement Idiom. This idiom models the uncertainty about the accuracy of a measurement instrument. This idiom emphasizes reasoning based on observations.
- The Induction Idiom. This idiom models the uncertainty related to inductive reasoning based on population of similar or exchangeable members. This idiom emphasizes statistical and analogical reasoning using historical data.
- The Reconciliation Idiom. This idiom models the reconciliation of results from competing measurement or prediction systems.

The flowchart in Figure 3 shows how to choose appropriate idioms.

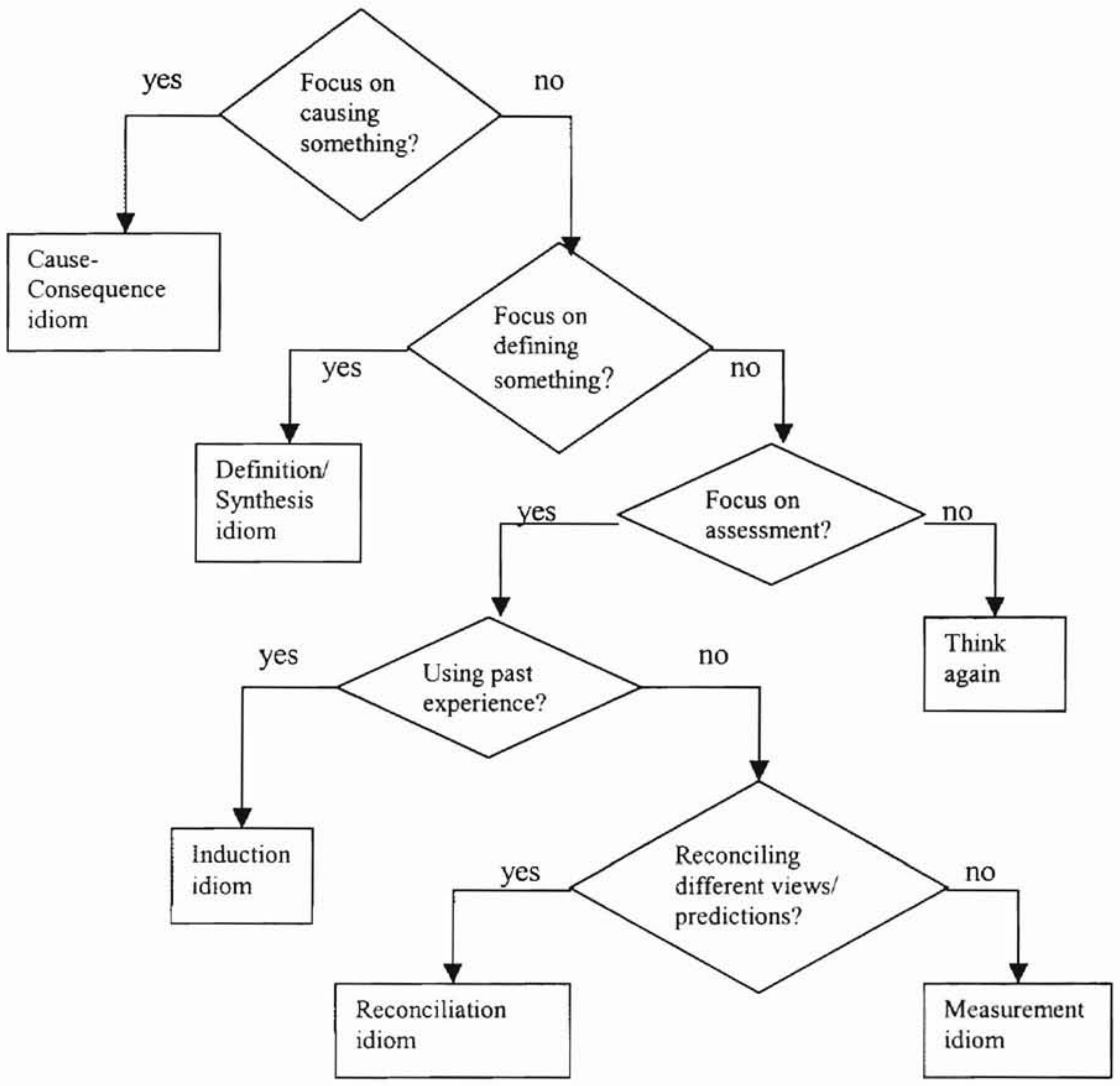


Figure 3. Choosing the Right Idiom [Neil et al. 99]

4.2.4.2 Determining an Appropriate Idiom for Each Group of Attributes

In this section, an appropriate idiom is determined for each group of attributes in Table VII.

For Group 1, the Measurement idiom was determined because software "Reliability" is measured (estimated or predicted) through "#latent fault per KLOC" in software and the consistency of software "Operational profile". The instantiation of this idiom, called reliability idiom, is given in Figure 4.

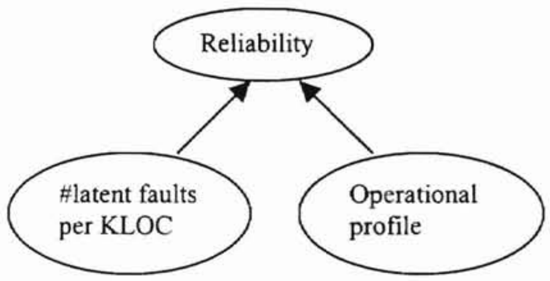


Figure 4. The Reliability Idiom

For Group 2, the Definition/Synthesis idiom was determined because "#latent faults per KLOC" is equal to "#faults introduced per KLOC" minus "#faults found per KLOC". The instantiation of this idiom, called latent faults idiom, is given in Figure 5.

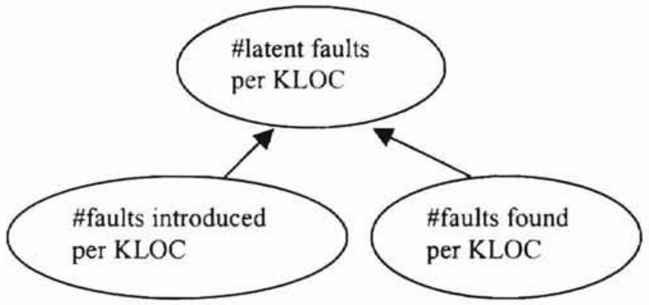


Figure 5. The Latent Faults Idiom

For Group 3, the Measurement idiom was determined since "#faults found per KLOC" is measured through "#faults introduced per KLOC" during software development and "Testing accuracy" during software testing process. The instantiation of this idiom, called faults found idiom, is given in Figure 6.

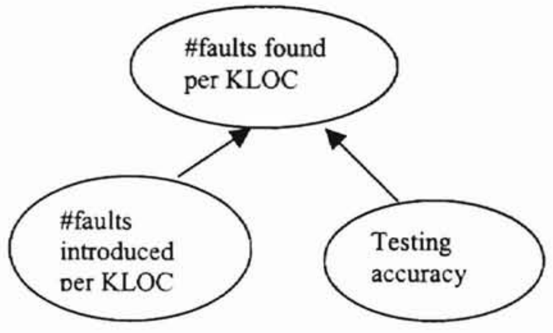


Figure 6. The Faults Found Idiom

For Group 4, the Measurement idiom was determined because "#faults introduced per KLOC" is measured through "Design quality" and "Coding accuracy" during software development. The higher the design quality is and the more accurate the coding process is, the less number of faults are introduced in software. The instantiation of this idiom, called faults introduced idiom, is given in Figure 7.

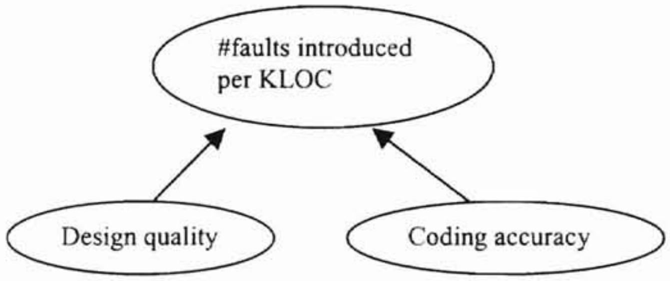


Figure 7. The Faults Introduced Idiom

For Group 5, the Definition/Synthesis idiom was determined since "Design quality" is defined by "Design experience", "Design effort", and "Problem complexity". The instantiation of this idiom, called design quality idiom, is shown in Figure 8.

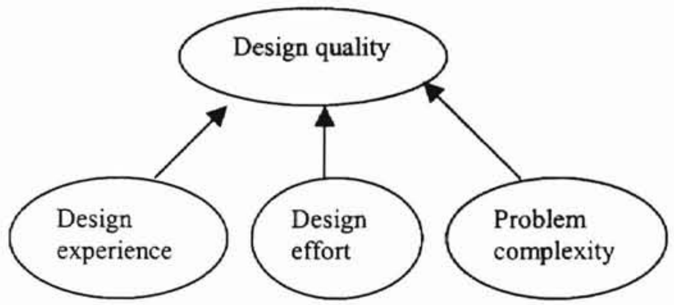


Figure 8. The Design Quality Idiom

For Group 6, the Definition/Synthesis idiom was determined because "Coding accuracy" is defined by "Coding experience" and "Coding effort". The instantiation of this idiom, called coding accuracy idiom, is given in Figure 9.

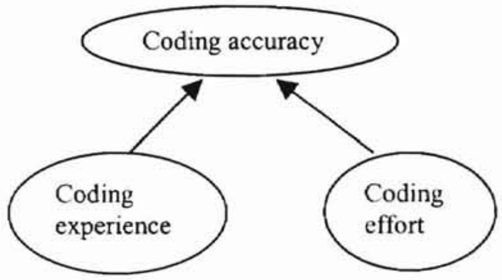


Figure 9. The Coding Accuracy Idiom

For Group 7, the Definition/Synthesis idiom was determined since "Testing accuracy" is defined by "Testing experience", "Testing effort", and "Problem complexity". The instantiation of this idiom, called testing accuracy idiom, is given in Figure 10.

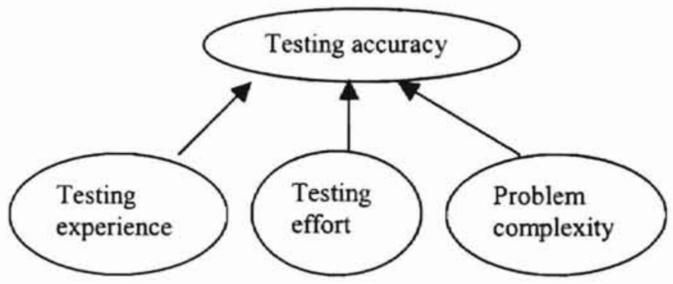


Figure 10. The Testing Accuracy Idiom

4.2.5 Defining Conditional Probability Tables (CPTs)

Seven idioms were determined in Section 4.2.4. A conditional probability table (CPT) for each node of the seven idioms is defined in this section. The SERENE 1.0 Demo tool was used to obtain all the CPTs. When defining a CPT, if a node had no parents, its probability is unconditional. The CPT for this node was a uniform distribution of probabilities over all the states of this node if the probabilities were not specified otherwise. That is, each state took a probability of $1/n$, where n is the number of states of the node. If a node had parents, its probability was conditional probability.

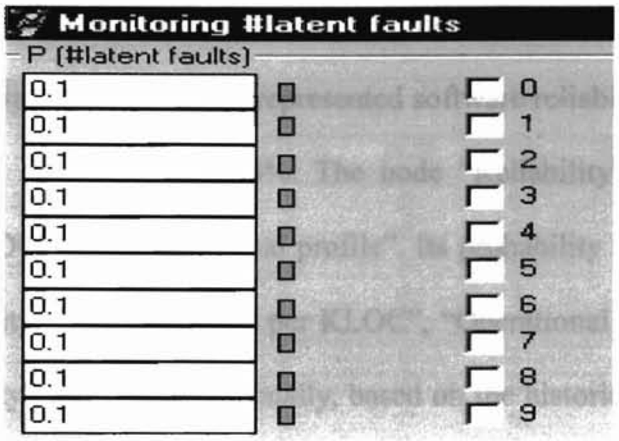
The SERENE 1.0 Demo tool provides two options for conditional probability specification: one is "Manually Specified Distributions", and the other is "Function Expressions". The option "Manually Specified Distributions" means that users need to assign each probability in the CPT manually based on historical data or expert knowledge. The option "Function Expressions" means that users need to supply an expression to indicate the relationship among the parent node(s) and the child node first, then the tool calculates the probabilities and defines the CPT for the child node based on the given expression and its parents' CPTs. Choosing an expression is based on historical data or expert knowledge. The expressions available in the SERENE 1.0 Demo tool include: constant values, arithmetic functions, Boolean functions, discrete distribution functions, and continuous distribution functions. The algorithms for calculating probabilities and defining CPT were given by Cowell et al. [Cowell et al. 99].

The rest of this section is organized as follows. In Section 4.2.5.1, the CPTs for all nodes of the reliability idiom are defined. In Section 4.2.5.2, the CPTs for all nodes of the latent faults idiom are defined. In Section 4.2.5.3, the CPTs for all nodes of the faults found idiom are defined. In Section 4.2.5.4, the CPTs for all nodes of the faults introduced idiom are defined. In Section 4.2.5.5, the CPTs for all nodes of the design quality idiom are defined. In Section 4.2.5.6, the CPTs for all nodes of the coding accuracy idiom are defined. In Section 4.2.5.7, the CPTs for all nodes of the testing accuracy idiom are defined.

4.2.5.1 CPTs for the Reliability Idiom

In the reliability idiom (Figure 4), there are three nodes, "#latent faults per KLOC", "Operational profile", and "Reliability".

The node "#latent faults per KLOC" was defined to have 10 discrete states: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The states represented software defect density (#defects/KLOC) as published [Neil and Fenton 96]. The node "#latent faults per KLOC" had no parents in the reliability idiom. Its CPT was uniform probability distribution. Each state took a probability of 0.1 as shown in Figure 11.

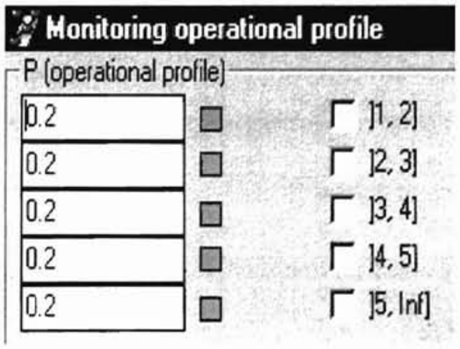

Monitoring #latent faults

P (#latent faults)

0.1	0
0.1	1
0.1	2
0.1	3
0.1	4
0.1	5
0.1	6
0.1	7
0.1	8
0.1	9

Figure 11. The CPT for "#latent faults per KLOC" in the Reliability Idiom

In order to be simple, five states were defined for the node "Operational profile". These five states were five continuous intervals: [1,2], [2,3], [3,4], [4,5], and [5,inf]. Here "inf" represents the last value in the last interval to follow the convention of the SERENE 1.0 Demo tool. All the "inf" in this thesis has the same meaning. The consistency of software operational profile was indicated by the changes of software operations. Interval [1,2] represented slightest changes or highest consistency of software operational profile. Interval [5,inf] represented most dramatic changes or lowest consistency of software operational profile. The node "Operational profile" had no parents. Its CPT was uniform probability distribution. Each state took a probability of 0.2 as given in Figure 12.


Monitoring operational profile

P (operational profile)

0.2	]1, 2]
0.2	]2, 3]
0.2	]3, 4]
0.2	]4, 5]
0.2	]5, Inf]

Figure 12. The CPT for "Operational profile"

The node "Reliability" was defined to have five continuous states: [5,6], [6,7], [7,8], [8,9], and [9,inf]. The values in each state represented software reliability – probability of failure-free operations, from 50% to 100%. The node "Reliability" had two parents: "#latent faults per KLOC" and "Operational profile". Its probability was the conditional probability P("Reliability" | "#latent faults per KLOC", "Operational profile"). The CPT for the node "Reliability" was defined manually, based on the historical data provided in [Neil and Fenton 96] and the understanding of the relationships among software reliability, number of latent faults, and software operational profile by people in OSU Bioinformatics lab. This CPT was given in Table VIII.

The particular values specified in the CPT of the node "Reliability" (Table VIII) reflected our belief that the less the "#latent faults per KLOC" is and the higher the consistency of software "Operational profile" is, the higher the "Reliability" of software is. The more the "#latent faults per KLOC" is and the less the consistency of software "Operational profile" is, the lower the "Reliability" of the software is. But "#latent faults per KLOC" had more impact on software reliability than the consistency of "Operational profile".

Table VIII. The CPT for "Reliability"

#latent faults	Operational profile	Reliability				
		[5,6]	[6,7]	[7,8]	[8,9]	[9,inf]
0	[1,2]	0	0	0	0	1
	[2,3]	0	0	0	0	1
	[3,4]	0	0	0	0	1
	[4,5]	0	0	0	0	1
	[5,inf]	0	0	0	0	1
1	[1,2]	0	0	0	0	1
	[2,3]	0	0	0	0	1
	[3,4]	0	0	0	0	1
	[4,5]	0	0	0	0.1	0.9
	[5,inf]	0	0	0.1	0.1	0.8
2	[1,2]	0	0	0	0	1
	[2,3]	0	0	0	0	1
	[3,4]	0	0	0	0.1	0.9
	[4,5]	0	0	0.1	0.1	0.8
	[5,inf]	0	0.1	0.1	0.1	0.7
3	[1,2]	0	0	0	0	1
	[2,3]	0	0	0	0.1	0.9
	[3,4]	0	0	0.1	0.1	0.8
	[4,5]	0	0.1	0.1	0.1	0.7
	[5,inf]	0.1	0.1	0.1	0.1	0.6
4	[1,2]	0	0	0	0.1	0.9
	[2,3]	0	0	0.1	0.1	0.8
	[3,4]	0	0.1	0.1	0.1	0.7
	[4,5]	0.1	0.1	0.1	0.1	0.6
	[5,inf]	0.1	0.1	0.1	0.2	0.5
5	[1,2]	0	0	0.1	0.1	0.8
	[2,3]	0	0.1	0.1	0.1	0.7
	[3,4]	0.1	0.1	0.1	0.1	0.6
	[4,5]	0.1	0.1	0.1	0.2	0.5
	[5,inf]	0.1	0.1	0.2	0.2	0.4
6	[1,2]	0	0.1	0.1	0.1	0.7
	[2,3]	0.1	0.1	0.1	0.1	0.6
	[3,4]	0.1	0.1	0.1	0.2	0.5
	[4,5]	0.1	0.1	0.2	0.2	0.4
	[5,inf]	0.1	0.2	0.2	0.2	0.3
7	[1,2]	0.1	0.1	0.1	0.1	0.6
	[2,3]	0.1	0.1	0.2	0.2	0.4
	[3,4]	0.2	0.2	0.2	0.2	0.2
	[4,5]	0.3	0.2	0.2	0.2	0.1
	[5,inf]	0.3	0.3	0.2	0.2	0
8	[1,2]	0.1	0.1	0.1	0.2	0.5
	[2,3]	0.1	0.1	0.2	0.2	0.4
	[3,4]	0.1	0.2	0.2	0.2	0.2
	[4,5]	0.3	0.3	0.3	0.1	0
	[5,inf]	0.5	0.3	0.2	0	0
9	[1,2]	0.1	0.1	0.2	0.2	0.4
	[2,3]	0.2	0.2	0.2	0.2	0.2
	[3,4]	0.4	0.3	0.2	0.1	0
	[4,5]	0.6	0.2	0.2	0	0
	[5,inf]	0.8	0.1	0.1	0	0

4.2.5.2 CPTs for the Latent Faults Idiom

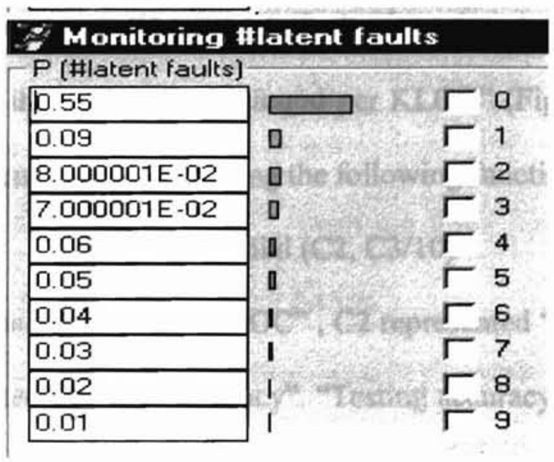
In the latent faults idiom (Figure 5), there are three nodes, "#faults introduced per KLOC", "#faults found per KLOC", and "#latent faults per KLOC".

All three nodes were defined to have the same states as the node "#latent faults per KLOC" in Section 4.2.5.1. The CPTs for the nodes "#faults introduced per KLOC" and "#faults found per KLOC" were both uniform probability distributions. Each state took a probability of 0.1.

The node "#latent faults per KLOC" had two parents: "#faults introduced per KLOC" and "#faults found per KLOC". Its probability was the conditional probability P("#latent faults per KLOC" | "#faults introduced per KLOC", "#faults found per KLOC"). The CPT for the node "#latent faults per KLOC" (Figure 13) was obtained from the SERENE 1.0 Demo tool by supplying the following function.

C1 = Max (0, C2 - C3)

where C1 represented "#latent faults per KLOC", C2 represented "#faults introduced per KLOC", and C3 represented "#faults found per KLOC".


Monitoring #latent faults

P (#latent faults)

0.55	0
0.09	1
8.000001E-02	2
7.000001E-02	3
0.06	4
0.05	5
0.04	6
0.03	7
0.02	8
0.01	9

Figure 13. The CPT for "#latent faults per KLOC" in the Latent Faults Idiom

In Figure 13, the probability of 0 latent fault was 55%, the probabilities of 1-9 latent faults decreased from 9% to 1% with 1% interval.

4.2.5.3 CPTs for the Faults Found Idiom

In the faults found idiom (Figure 6), there are three nodes, "#faults introduced per KLOC", "Testing accuracy" and "#faults found per LOC".

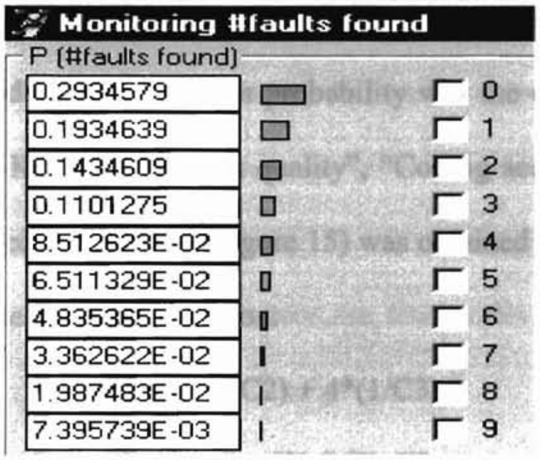
The node "#faults introduced per KLOC" had exactly the same states and CPT as it in the latent faults idiom in Section 4.2.5.2.

The node "Testing accuracy" was defined to have ten continuous states: [0,1], [1,2], [2,3], [3,4], [4,5], [5,6], [6,7], [7,8], [8,9], and [9,inf]. Interval [0,1] represented the lowest testing accuracy. Interval [9,inf] represented the highest testing accuracy. The CPT for the node "Testing accuracy" was uniform probability distribution. Each state took a probability of 0.1.

The node "#faults found per KLOC" had the same states as it in the latent faults idiom in Section 4.2.5.2. The node "#faults found per KLOC" had two parents: "#faults introduced per KLOC" and "Testing accuracy". Its probability was the conditional probability P("#faults found per KLOC" | "#faults introduced per KLOC", "Testing accuracy"). The CPT for the node "#faults found per KLOC" (Figure 14) was obtained from the SERENE 1.0 Demo tool by supplying the following function.

$$C1 = \text{Binomial}(C2, C3/10)$$

where C1 represented "#faults found per KLOC" , C2 represented "#faults introduced per KLOC", and C3 represented "Testing accuracy". "Testing accuracy"/10 is the probability of finding faults in software. When testing accuracy was very high (in state [9,inf]), the value of "Testing accuracy"/10, or the probability of finding faults would be close to 1. When testing accuracy was very low (in state [0,1]), the value of "Testing accuracy"/10, or the probability of finding faults would be close to 0.



P (#faults found)

0.2934579	0
0.1934639	1
0.1434609	2
0.1101275	3
8.512623E-02	4
6.511329E-02	5
4.835365E-02	6
3.362622E-02	7
1.987483E-02	8
7.395739E-03	9

Figure 14. The CPT for "#faults found per KLOC"

In Figure 14, the probability of 0 fault found was 29%, the probability of 1 fault found was 19%, the probability of 2 faults found was 14%, and the probability of 3 faults found was 11%. The probabilities of 4-8 faults found decreased from 8.5% to 2.0% with about 2% interval. The probability of 9 faults found was 0.7%.

4.2.5.4 CPTs for the Faults Introduced Idiom

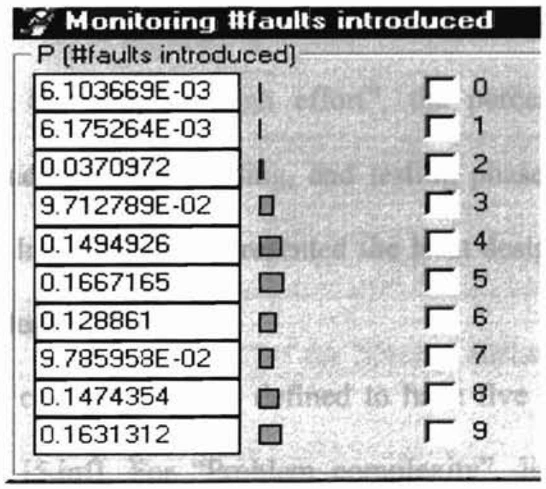
In the faults introduced idiom (Figure 7), there are three nodes, "Design quality", "Coding accuracy", and "#faults introduced per KLOC".

The nodes "Design quality" and "Coding accuracy" were defined to have five continuous states: [0,1], [1,2], [2,3], [3,4], and [4,inf]. Interval [0,1] represented very poor design quality or very low coding accuracy. Interval [4,inf] represented very high design quality or very high coding accuracy. The CPTs for the nodes "Design quality" and "Coding accuracy" were both uniform probability distributions. Each state took a probability of 0.2.

The node "#faults introduced per KLOC" had the same states as it in the faults found idiom in Section 4.2.5.3. The node "#faults introduced per KLOC" had two parents: "Design quality" and "Coding accuracy". Its probability was the conditional probability P("#faults introduced per KLOC" | "Design quality", "Coding accuracy"). The CPT for the node "#faults introduced per KLOC" (Figure 15) was obtained from the SERENE 1.0 Demo tool by supplying the following function.

$$C1 = 6*(1/C2) + 4*(1/C3)$$

where C1 represented "#faults introduced per KLOC", C2 represented "Design quality", and C3 represented "Coding accuracy".



P (#faults introduced)

6.103669E-03	0
6.175264E-03	1
0.0370972	2
9.712789E-02	3
0.1494926	4
0.1667165	5
0.128861	6
9.785958E-02	7
0.1474354	8
0.1631312	9

Figure 15. The CPT for "#faults introduced per KLOC"

In Figure 15, the probability of 0 or 1 fault introduced was 0.6%, respectively, the probability of 2 faults introduced was 4%, the probability of 3 faults introduced was 10%, the probabilities of 4 faults introduced was 15%, the probability of 5 faults introduced was 17%, the probability of 6 faults introduced was 13%, the probability of 7 faults introduced was 10%, the probability of 8 faults introduced was 15%, and the probability of 9 faults introduced was 16%.

4.2.5.5 CPTs for the Design Quality Idiom

In the design quality idiom (Figure 8), there are four nodes, "Design experience", "Design effort", "Problem complexity", and "Design quality".

The nodes "Design experience" and "Design effort" were both defined to have five continuous states: [0,1], [1,2], [2,3], [3,4], and [4,inf]. For the node "Design experience", years of design were used to indicate the levels of different design experience. Interval [0,1] represented the least design experience. Interval [4,inf] represented the richest design experience. For the node "Design effort", the percentage of total software development time (including design, coding, and testing phases) was used to indicate different design efforts. Interval [0,1] represented the least design effort. Interval [4,inf] represented the highest design effort.

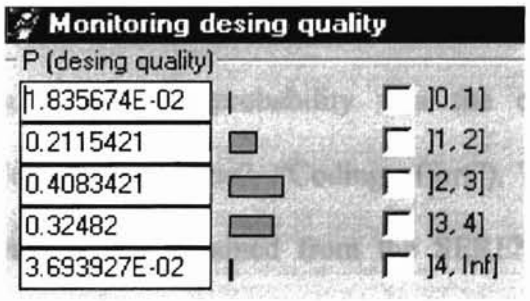
The node "Problem complexity" was defined to have five continuous states: [1,2], [2,3], [3,4], [4,5], and [5,inf]. For "Problem complexity", lines of codes (LOC) in software were used to indicate different levels of problem complexity. Since LOC cannot be 0, intervals started from 1. Interval [1,2] represented the most difficult problem. Interval [5,inf] represented the easiest problem.

The CPTs for the nodes "Design experience", "Design effort", and "Problem complexity" were all uniform probability distributions. Each state took a probability of 0.2.

The node "Design quality" took the same states as it in the faults introduced idiom in Section 4.2.5.4. The node "Design quality" had three parents: "Design experience", "Design effort", and "Problem complexity". Its probability was the conditional probability P("Design Quality" | "Design experience", "Design effort", "Problem complexity"). The CPT for "Design quality" (Figure 16) was obtained from the SERENE 1.0 Demo tool by supplying the following function.

$$C1 = 2/4*C2 + 1/4*C3 + 1/4*C4$$

where C1 represented "Design quality", C2 represented "Design experience", C3 represented "Design effort", and C4 represented "Problem complexity".


Monitoring desing quality

P (desing quality)

1.835674E-02	]0, 1]
0.2115421	]1, 2]
0.4083421	]2, 3]
0.32482	]3, 4]
3.693927E-02	]4, Inf]

Figure 16. The CPT for "Design quality"

In Figure 16, the probability of design quality at very low level (state [0,1]) was 2%, the probability of design quality at low level (state [1,2]) was 21%, the probability of design quality at medium level (state [2,3]) was 41%, the probability of design quality at high level (state [3,4]) was 32%, and the probability of design quality at very high level was 4%.

4.2.5.6 CPTs for the Coding Accuracy Idiom

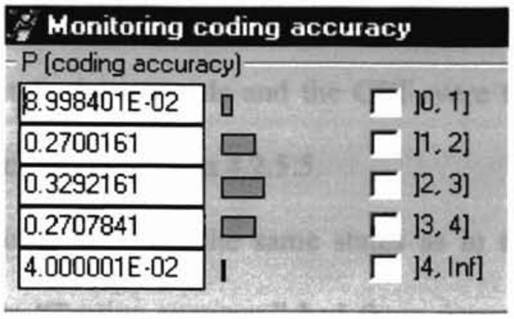
In the coding accuracy idiom (Figure 9), there are three nodes, "Coding experience", "Coding effort", and "Coding accuracy".

The nodes "Coding experience" and "Coding effort" were both defined to have five continuous states: [0,1], [1,2], [2,3], [3,4], and [4,inf]. For the nodes "Coding experience" and "Coding effort", the interpretation of intervals was the same as for the nodes "Design experience" and "Design effort" in Section 4.2.5.5. The CPTs for the nodes "Coding experience" and "Coding effort" were both uniform probability distributions. Each state took a probability of 0.2.

The node "Coding accuracy" took the same states as it did in the faults introduced idiom in Section 4.2.5.4. The node "Coding accuracy" had two parents: "Coding experience" and "Coding effort". Its probability was the conditional probability P("Coding accuracy" | "Coding experience", "Coding effort"). The CPT for the node "Coding accuracy" (Figure 17) was obtained from the SERENE 1.0 Demo tool by supplying the follow function.

$$C1 = 2/3*C2 + 1/3*C3$$

where C1 represented "Coding accuracy", C2 represented "Coding experience", and C3 represented "Coding effort".



Monitoring coding accuracy

P (coding accuracy)

8.998401E-02	]0, 1]
0.2700161	]1, 2]
0.3292161	]2, 3]
0.2707841	]3, 4]
4.000001E-02	]4, Inf]

Figure 17. The CPT for "Coding accuracy"

In Figure 17, the probability of coding accuracy at very low level (state [0,1]) was 9%, the probability of coding accuracy at low level (state [1,2]) was 27%, the probability of coding accuracy at medium level (state [2,3]) was 33%, the probability of coding accuracy at high level (state [3,4]) was 27%, and the probability of coding accuracy at very high level was 4%.

4.2.5.7 CPTs for the Testing Accuracy Idiom

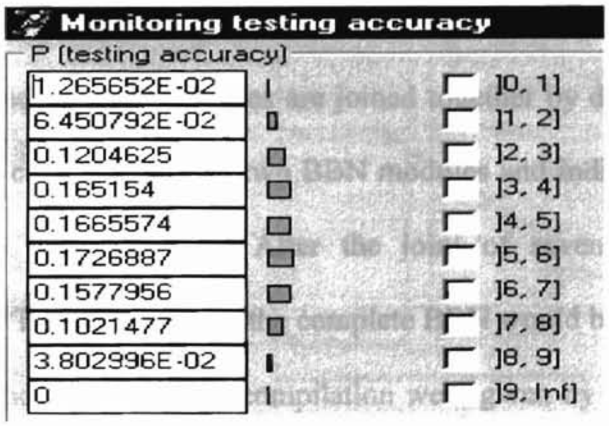
In the testing accuracy idiom (Figure 10), there are four nodes, "Testing experience", "Testing effort", "Problem complexity", and "Testing accuracy".

The nodes "Testing experience" and "Testing effort" were both defined to have ten continuous states: [0,1], [1,2], [2,3], [3,4], [4,5], [5,6], [6,7], [7,8], [8,9], and [9,inf]. For the node "Testing experience", years of testing were used to indicate the different levels of testing experience. Interval [0,1] represented the least testing experience. Interval [9,inf] represented the richest testing experience. For the node "Testing effort", the percentage of total software development time was used to indicate different testing effort. Interval [0,1] represented the least testing effort. Interval [9,inf] represented the highest testing effort. The CPTs for the nodes "Testing experience" and "Testing effort" were both uniform probability distributions. Each state took a probability of 0.1. For the node "Problem complexity", the intervals and the CPT were the same as intervals and CPT for "Problem complexity" in Section 4.2.5.5.

The node "Testing accuracy" took the same states as in the faults found idiom in Section 4.2.5.3. The node "Testing accuracy" had three parents: "Testing experience", "Testing effort", and "Problem complexity". Its probability was the conditional probability P("Testing accuracy" | "Testing experience", "Testing effort", "Problem complexity"). The CPT for the node "Testing Accuracy" (Figure 18) was obtained from the SERENE 1.0 Demo tool by supplying the following function.

$$C1 = 6/10*C2 + 3/10*C3 + 1/10*C4$$

where C1 represented "Testing Accuracy", C2 represented "Testing experience", C3 represented "Testing effort", and C4 represented "Problem complexity".


Monitoring testing accuracy

P (testing accuracy)

1.265652E-02	]0, 1]
6.450792E-02	]1, 2]
0.1204625	]2, 3]
0.165154	]3, 4]
0.1665574	]4, 5]
0.1726887	]5, 6]
0.1577956	]6, 7]
0.1021477	]7, 8]
3.802996E-02	]8, 9]
0	]9, Inf]

Figure 18. The CPT for "Testing accuracy"

In Figure 18, the probability of testing accuracy at very low level (states [0,1] and [1,2]) was 8%, the probability of testing accuracy at low level (states [2,3] [3,4]) was

28%, the probability of testing accuracy at medium level (states [4,5] and [5,6]) was 34%, the probability of testing accuracy at high level (states [6,7] and [7,8]) was 26%, and the probability of testing accuracy at very high level (states [8,9] and [9,inf]) was 4%.

4.2.6. Building Complete Reliability BBN

In Section 4.2.4, seven idioms were determined. In Section 4.2.5, the CPTs for each node of the seven idioms were defined. An idiom with the CPT of each of its nodes defined is called a BBN module. In different BBN modules, nodes with the same name and the same states are called common node of these BBN modules. Different BBN modules can be joined together through common nodes. If no common node exists in two BBN modules, a virtual common node can be introduced to join the two BBN modules. In this section, seven BBN modules were joined together by common nodes. The complete BBN is given in Figure 19.

In Figure 19, the seven BBN modules are joined together by double arrows. Double arrows connected the common node in two BBN modules and indicated the direction of joining of each two BBN modules. After the joint of seven BBN modules, the probabilities in the CPT of each node in the complete BBN would be revised by a process called compilation. The algorithms for compilation were given by Cowell et al. [Cowell et al. 99]. After the compilation, the complete BBN will be ready to make predictions.

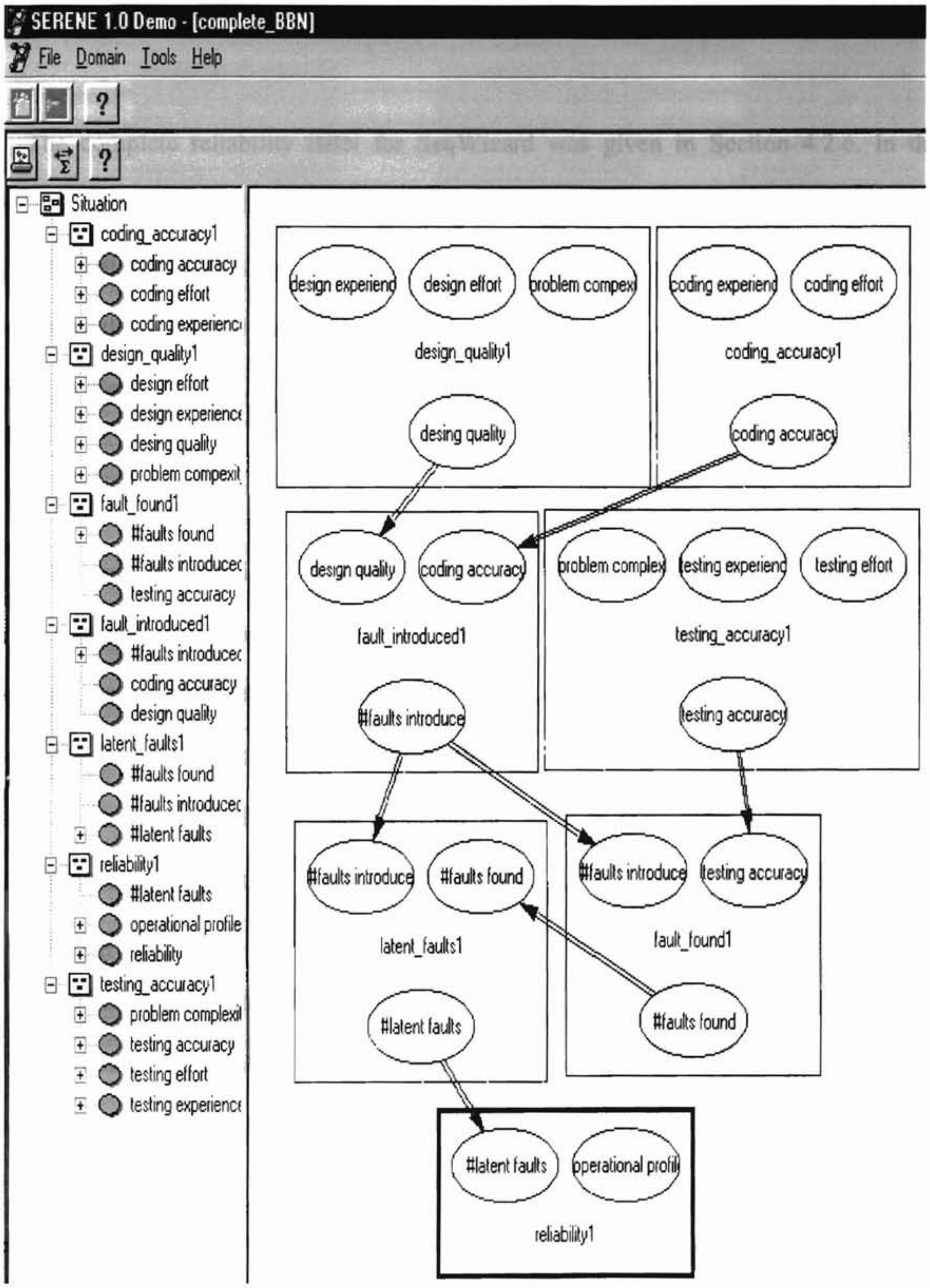


Figure 19. The Complete Reliability BBN for SeqWizard

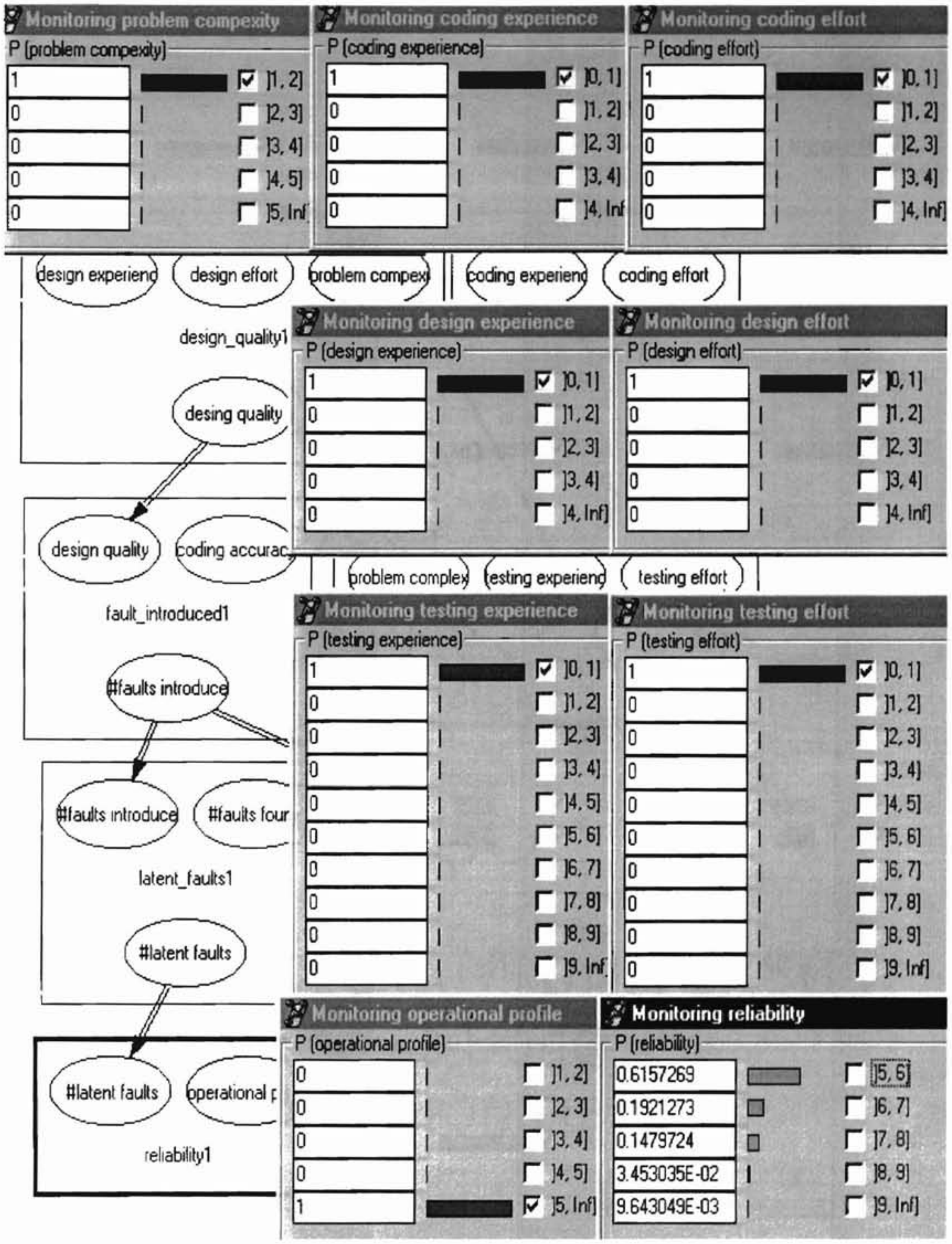
4.2.7 Predicting Reliability

The complete reliability BBN for SeqWizard was given in Section 4.2.6. In this section, the reliability BBN is used to predict the reliability of SeqWizard. A set of hypothetical data was also entered to the reliability BBN to check its suitability before predicting the reliability of SeqWizard. Checking the suitability of the reliability BBN is given in Section 4.2.7.1. Predicting the reliability of SeqWizard is given in Section 4.2.7.2.

4.2.7.1 Checking the Suitability of the Reliability BBN

In this section, the worst case scenario (Figure 20), the average case scenario (Figure 21), and the best case scenario (Figure 22) are generated to check the suitability of the reliability BBN. For each case, the hypothetical data were entered to the corresponding nodes in the reliability BBN, then the probability in each state of the node "Reliability" was observed after a process called propagation. The algorithms for propagation were given by Cowell et al. [Cowell et al. 99].

As defined before, the node "Reliability" had five states: [5,6], [6,7], [7,8], [8,9], and [9,inf]. These states represented software reliability – probability of failure-free operations, from 50% to 100%. Informally, these states could be interpreted as "Very Low", "Low", "Medium", "High", and "Very High" reliability, respectively.



Monitoring problem compexity

P (problem compexity)

1	✓]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, 5]
0	]5, Inf

Monitoring coding experience

P (coding experience)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, Inf

Monitoring coding effort

P (coding effort)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, Inf]

Monitoring design experience

P (design experience)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, Inf]

Monitoring design effort

P (design effort)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, Inf]

Monitoring testing experience

P (testing experience)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, 5]
0	]5, 6]
0	]6, 7]
0	]7, 8]
0	]8, 9]
0	]9, Inf

Monitoring testing effort

P (testing effort)

1	✓]0, 1]
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, 5]
0	]5, 6]
0	]6, 7]
0	]7, 8]
0	]8, 9]
0	]9, Inf]

Monitoring operational profile

P (operational profile)

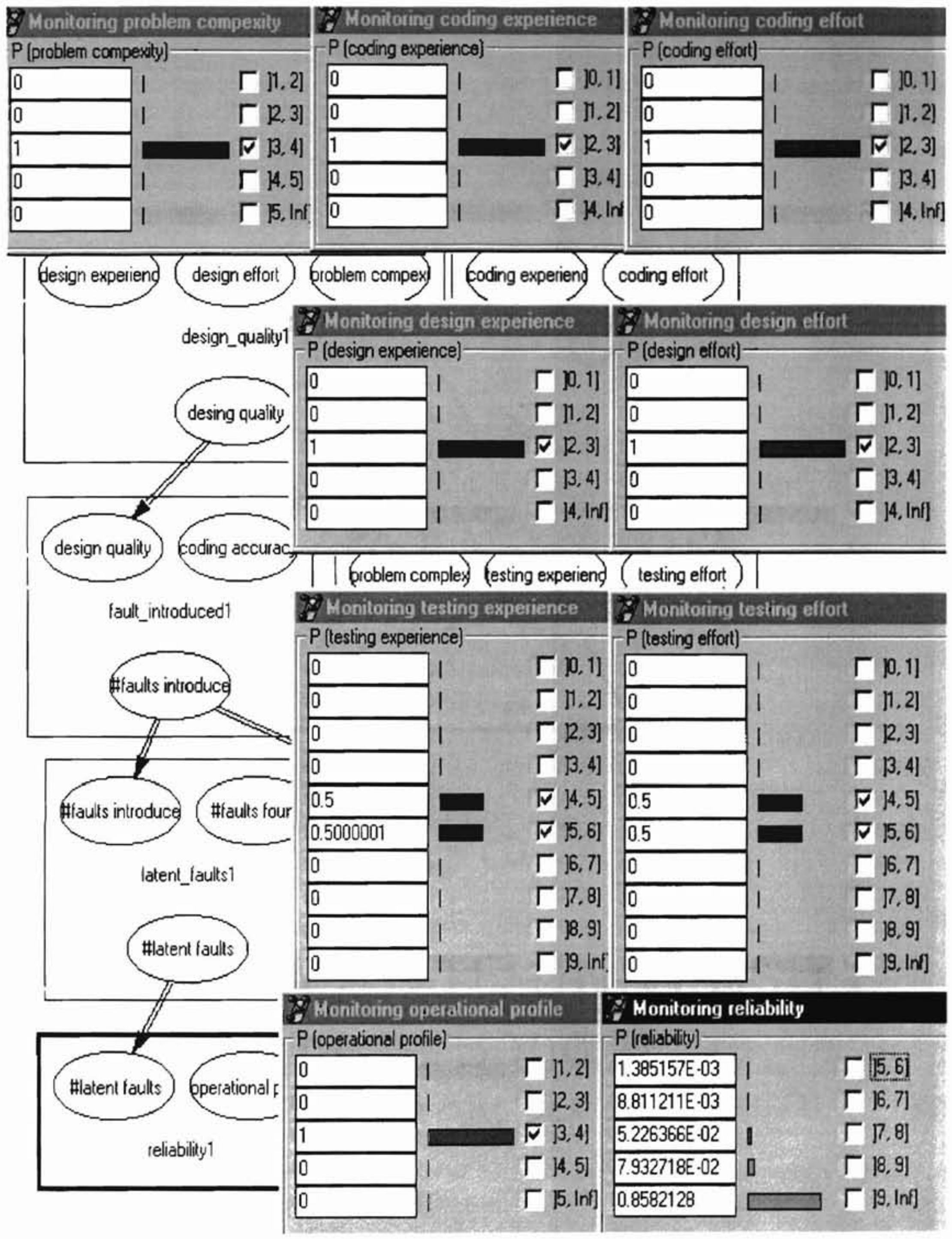
0	]1, 2]
0	]2, 3]
0	]3, 4]
0	]4, 5]
1	✓]5, Inf]

Monitoring reliability

P (reliability)

0.6157269	]5, 6]
0.1921273	]6, 7]
0.1479724	]7, 8]
3.453035E-02	]8, 9]
9.643049E-03	]9, Inf]

Figure 20. The Worst Case Scenario of the Reliability BBN for SeqWizard



Monitoring problem compexity

P (problem compexity)

0		]1, 2]
0		]2, 3]
1	✓	]3, 4]
0		]4, 5]
0		]5, Inf

Monitoring coding experience

P (coding experience)

0		]0, 1]
0		]1, 2]
1	✓	]2, 3]
0		]3, 4]
0		]4, Inf

Monitoring coding effort

P (coding effort)

0		]0, 1]
0		]1, 2]
1	✓	]2, 3]
0		]3, 4]
0		]4, Inf]

Monitoring design experience

P (design experience)

0		]0, 1]
0		]1, 2]
1	✓	]2, 3]
0		]3, 4]
0		]4, Inf]

Monitoring design effort

P (design effort)

0		]0, 1]
0		]1, 2]
1	✓	]2, 3]
0		]3, 4]
0		]4, Inf]

Monitoring testing experience

P (testing experience)

0		]0, 1]
0		]1, 2]
0		]2, 3]
0		]3, 4]
0.5	✓	]4, 5]
0.5000001	✓	]5, 6]
0		]6, 7]
0		]7, 8]
0		]8, 9]
0		]9, Inf

Monitoring testing effort

P (testing effort)

0		]0, 1]
0		]1, 2]
0		]2, 3]
0		]3, 4]
0.5	✓	]4, 5]
0.5	✓	]5, 6]
0		]6, 7]
0		]7, 8]
0		]8, 9]
0		]9, Inf]

Monitoring operational profile

P (operational profile)

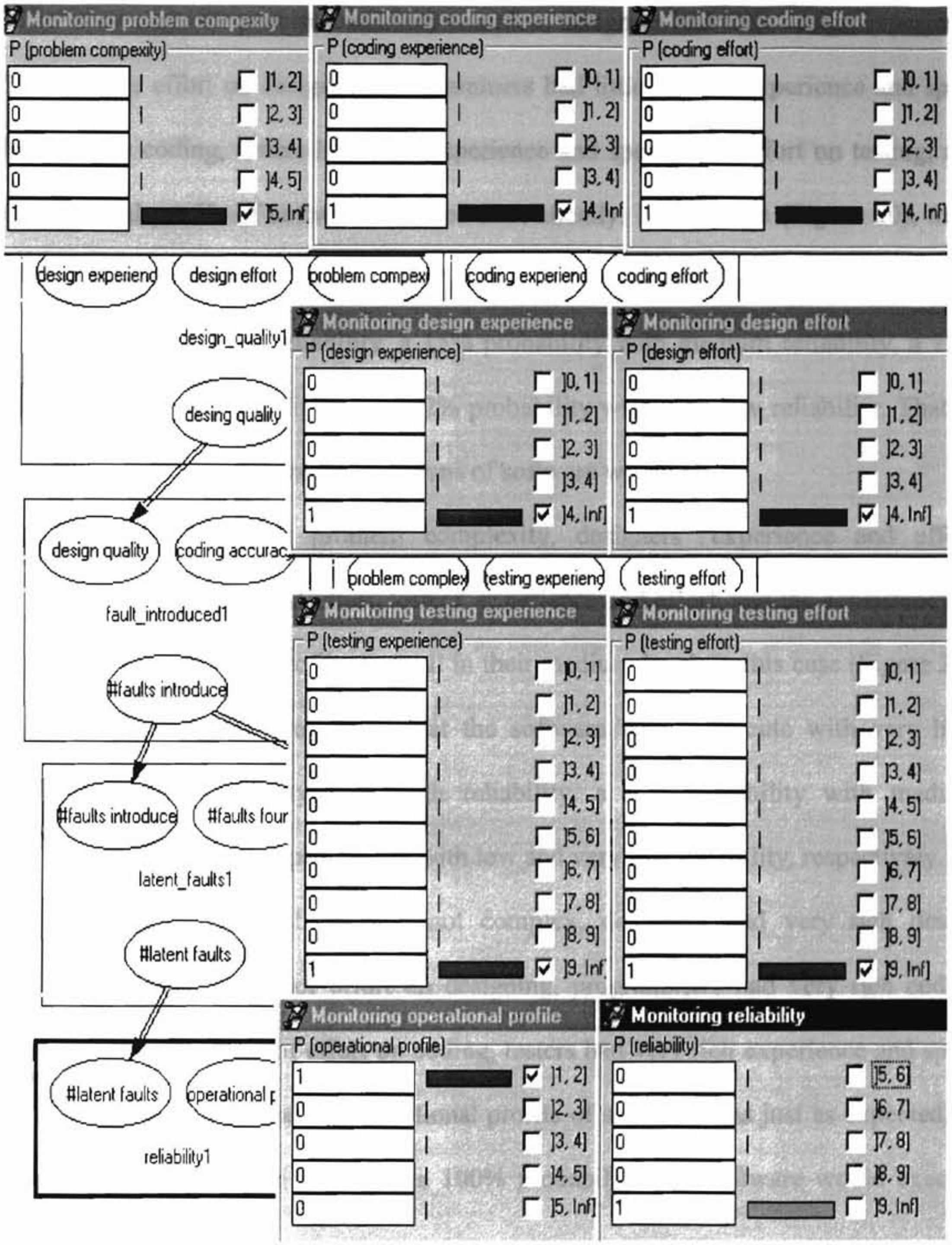
0		]1, 2]
0		]2, 3]
1	✓	]3, 4]
0		]4, 5]
0		]5, Inf]

Monitoring reliability

P (reliability)

1.385157E-03		]5, 6]
8.811211E-03		]6, 7]
5.226366E-02		]7, 8]
7.932718E-02		]8, 9]
0.8582128		]9, Inf]

Figure 21. The Average Case Scenario of the Reliability BBN for SeqWizard



Monitoring problem compexity

P (problem compexity)

0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
0	☐	]4, 5]
1	✓	]5, Inf]

Monitoring coding experience

P (coding experience)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
1	✓	]4, Inf]

Monitoring coding effort

P (coding effort)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
1	✓	]4, Inf]

Monitoring design experience

P (design experience)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
1	✓	]4, Inf]

Monitoring design effort

P (design effort)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
1	✓	]4, Inf]

Monitoring testing experience

P (testing experience)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
0	☐	]4, 5]
0	☐	]5, 6]
0	☐	]6, 7]
0	☐	]7, 8]
0	☐	]8, 9]
1	✓	]9, Inf]

Monitoring testing effort

P (testing effort)

0	☐	]0, 1]
0	☐	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
0	☐	]4, 5]
0	☐	]5, 6]
0	☐	]6, 7]
0	☐	]7, 8]
0	☐	]8, 9]
1	✓	]9, Inf]

Monitoring operational profile

P (operational profile)

1	✓	]1, 2]
0	☐	]2, 3]
0	☐	]3, 4]
0	☐	]4, 5]
0	☐	]5, Inf]

Monitoring reliability

P (reliability)

0	☐	]5, 6]
0	☐	]6, 7]
0	☐	]7, 8]
0	☐	]8, 9]
1	☐	]9, Inf]

Figure 22. The Best Case Scenario of the Reliability BBN for SeqWizard

The worst case: the problem was very complex, designers had little design experience and spent little effort on designing, programmers had little coding experience and spent little effort on coding, testers had little experience and spent little effort on testing, and the operational profile of software changed dramatically. In this case (Figure 20), there would be a 1% probability that the software would execute with very high reliability, a 3% probability with high reliability, a 15% probability with medium reliability, a 19% probability with low reliability, and a 62% probability with very low reliability. That is, most of the time, about half of the total runs of software would fail.

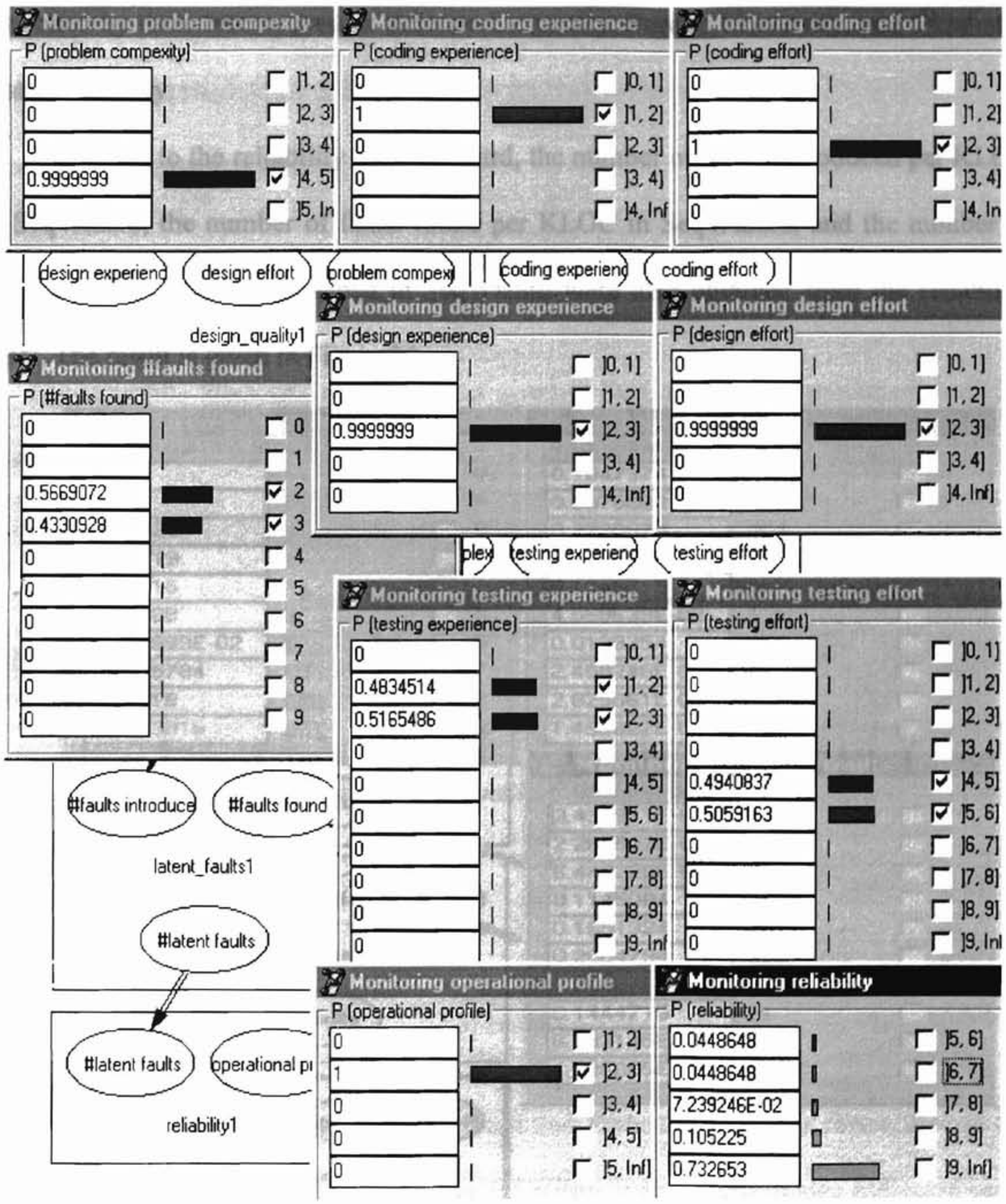
The average case: the problem complexity, designers' experience and effort, programmers' experience and effort, testers' experience and effort, and the consistency of the software's operational profile were all in their medium level. In this case (Figure 21), there would be an 86% probability that the software would execute with very high reliability, a 8% probability with high reliability, a 5% probability with medium reliability, a 0.9% and 0.1% probability with low and very low reliability, respectively.

The best case: the problem was not complex, designers had very rich design experience and spent a lot of effort on designing, programmers had very rich coding experience and spent a lot of effort on coding, testers had very rich experience and spent a lot of effort on testing, and the operational profile of software was just as expected. In this case (Figure 22), there would be a 100% probability that software would execute with very high reliability.

The results of the worst case scenario, the average case scenario, and the best case scenario were in accordance with the reliability cost function [Lyu 96]. This indicated that the reliability BBN constructed was suitable for software reliability prediction.

4.2.7.2 Predicting the Reliability of SeqWizard

The measurements and observations collected during SeqWizard development (Table VI) were entered in the corresponding nodes in the reliability BBN. The propagation result is given in Figure 23 below.

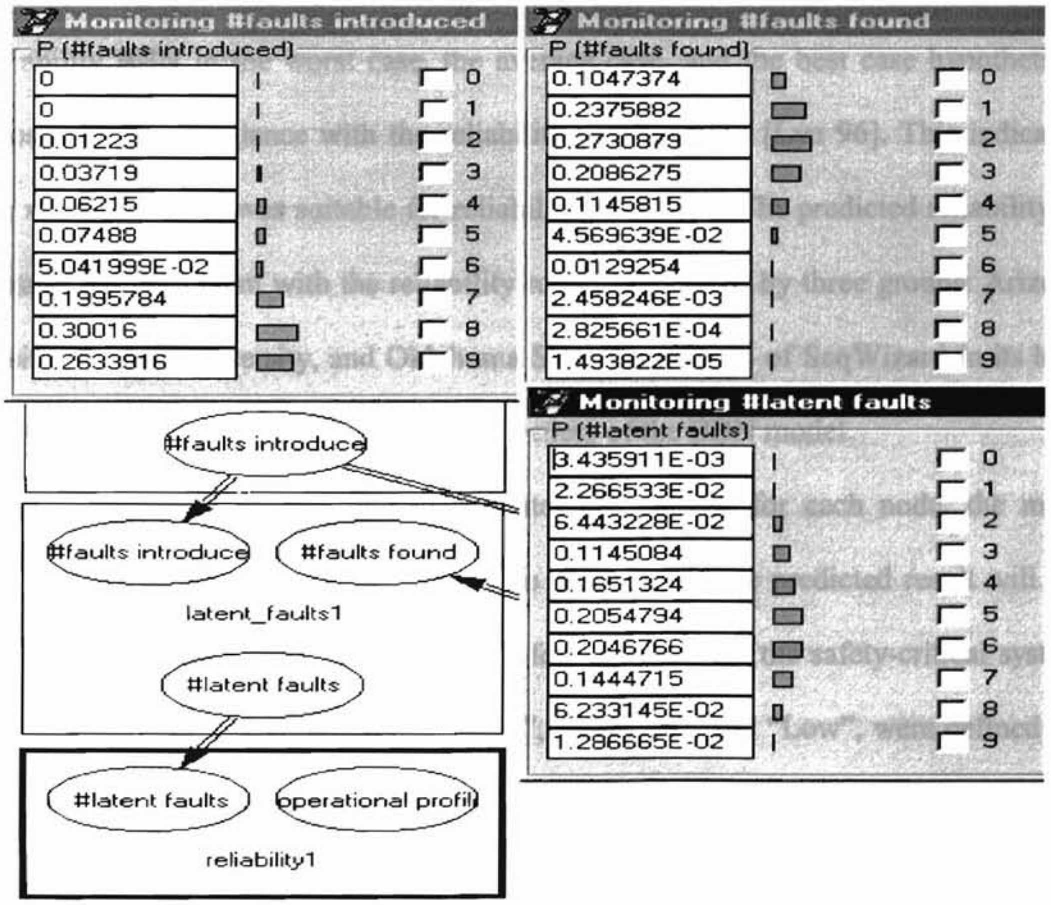


```
Monitoring problem compexity
P (problem compexity)
0           ]1, 2]
0           ]2, 3]
0           ]3, 4]
0.9999999   [✓ ]4, 5]]
0           ]5, In

Monitoring coding experience
P (coding experience)
0           ]0, 1]
1           [✓ ]1, 2]]
0           ]2, 3]
0           ]3, 4]
0           ]4, Inf

Monitoring coding effort
P (coding effort)
0           ]0, 1]
0           ]1, 2]
1           [✓ ]2, 3]]
0           ]3, 4]
0           ]4, In

Monitoring design experience
P (design experience)
0           ]0, 1]
0           ]1, 2]
0.9999999   [✓ ]2, 3]]
0           ]3, 4]
0           ]4, Inf]

Monitoring design effort
P (design effort)
0           ]0, 1]
0           ]1, 2]
0.9999999   [✓ ]2, 3]]
0           ]3, 4]
0           ]4, Inf]

Monitoring #faults found
P (#faults found)
0           0
0           1
0.5669072   [✓ 2]
0.4330928   [✓ 3]
0           4
0           5
0           6
0           7
0           8
0           9

Monitoring testing experience
P (testing experience)
0           ]0, 1]
0.4834514   [✓ ]1, 2]]
0.5165486   [✓ ]2, 3]]
0           ]3, 4]
0           ]4, 5]
0           ]5, 6]
0           ]6, 7]
0           ]7, 8]
0           ]8, 9]
0           ]9, Inf

Monitoring testing effort
P (testing effort)
0           ]0, 1]
0           ]1, 2]
0           ]2, 3]
0           ]3, 4]
0.4940837   [✓ ]4, 5]]
0.5059163   [✓ ]5, 6]]
0           ]6, 7]
0           ]7, 8]
0           ]8, 9]
0           ]9, Inl

Monitoring operational profile
P (operational profile)
0           ]1, 2]
1           [✓ ]2, 3]]
0           ]3, 4]
0           ]4, 5]
0           ]5, Inf]

Monitoring reliability
P (reliability)
0.0448648      ]5, 6]
0.0448648      ]6, 7]
7.239246E-02   ]7, 8]
0.105225       ]8, 9]
0.732653       ]9, Inf]
```

Figure 23. Predicted Reliability of SeqWizard

In Figure 23, there is a 73.3% probability that SeqWizard would execute with very high reliability, a 10.5% probability with high reliability, a 7.2% probability with medium reliability, a 4.5 % probability with low or very low reliability. This result is very close to the reliability report during the beta testing of SeqWizard. The reliability report was generated by three testing groups: Arizona University, Purdue University, and Oklahoma State University.

In addition to the reliability of SeqWizard, the number of faults introduced per KLOC to SeqWizard, the number of faults found per KLOC in SeqWizard, and the number of latent faults per KLOC remained in SeqWizard were also obtained from the reliability BBN. The result is given in Figure 24.



Monitoring #faults introduced

P (#faults introduced)	
0	0
0	1
0.01223	2
0.03719	3
0.06215	4
0.07488	5
5.041999E-02	6
0.1995784	7
0.30016	8
0.2633916	9

Monitoring #faults found

P (#faults found)	
0.1047374	0
0.2375882	1
0.2730879	2
0.2086275	3
0.1145815	4
4.569639E-02	5
0.0129254	6
2.458246E-03	7
2.825661E-04	8
1.493822E-05	9

Monitoring #latent faults

P (#latent faults)	
3.435911E-03	0
2.266533E-02	1
6.443228E-02	2
0.1145084	3
0.1651324	4
0.2054794	5
0.2046766	6
0.1444715	7
6.233145E-02	8
1.286665E-02	9

Figure 24. Predicted Faults Introduced, Faults Found, and Latent Faults in SeqWizard

In Figure 24, in the "Monitoring #faults introduced" window, the probability of 7 or more faults per KLOC introduced in SeqWizard is more than 76% (26% + 30% + 20%). In the "Monitoring #faults found" window, the probability of finding 1-3 faults in SeqWizard is more than 71% (23%+27%+21%). In the "Monitoring #latent faults" window, the probability of 5 or more latent faults per KLOC remained in SeqWizard is more than 62% (1% + 6% + 14% + 20.5% + 20.5%).

4.3 Discussion

The reliability BBN of SeqWizard was given in Section 4.2. The predicted results of this reliability BBN in the worst case, the average case, and the best case hypothetical scenarios were in accordance with the reliability cost function [Lyu 96]. This indicated that the reliability BBN was suitable for reliability prediction. The predicted reliability of SeqWizard was consistent with the reliability report (generated by three groups: Arizona University, Purdue University, and Oklahoma State University) of SeqWizard in its beta testing stage. This result confirmed the effectiveness of the BBN model.

When constructing a BBN, the more states are defined for each node, the more complex the BBN model is and the higher the precision of the predicted result will be. Two states, "good" and "poor", were defined for each node of the safety-critical system BBN [Bouissou et al. 99]. Three states, "High", "Medium", and "Low", were defined for each node of the software quality BBN [Neil and Fenton 96]. In this thesis, five to ten states were defined for each node of the reliability BBN. These states indicated some intermediate levels between the two extremes.

The effectiveness of the BBN model for predictive analysis was confirmed by this thesis. But constructing a suitable BBN is a complex process. Probably the most difficult part was defining the conditional probability table (CPT) for each node in the BBN. The difficulty came from three limitations: 1) no standards available; 2) lack of historical data (many software systems have been developed and deployed, but few measurements and observations are available); and 3) uncertain nature of attributes. These limitations are the major reasons why the BBN model is not widely applied in software engineering [Fenton and Neil 99]. The other reasons include the fact that the probabilistic explanation of events is abstract [Cowell et al. 99].

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

In this thesis, a survey of six current and popular software reliability models was provided and one of the six models, the BBN model, was analyzed in detail.

For the six software reliability models investigated, the following issues were discussed for each model: a brief history, model classification, software development phase(s) applied, basic assumptions, data requirements, model form, and application scope. The six models were also compared from three aspects: factors modeled, availability of CASE tools, and advantages and disadvantages. Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, and Shooman's model model two kinds of factors: 1) software failure data obtained in software test and operation processes, and 2) competence of software testers. These four models mainly focus on reliability estimation. The advantages of these four models are their simplicity and low cost to implement. The disadvantages are: 1) some assumptions are too simple; 2) some reliability causal factors are not modeled; and 3) they cannot handle uncertainty. The Littlewood-Verral model and the Bayesian Belief Networks (BBN) model model factors in each phase of the software life cycle when failure data may or may not be available.

These two models focus on reliability prediction and estimation. The advantages of these two models are: 1) they are based on practical assumptions; 2) they model most reliability causal factors; and 3) they can handle uncertainty. The disadvantages are their complex model forms and difficulty in implementation. For all the six models, CASE tools were also investigated in detail. These survey results can be used by practitioners to help make a decision when choosing among software reliability models.

The reliability BBN for SeqWizard was constructed through seven steps in order to analyze it in detail. The suitability of this BBN was checked by a set of hypothetical scenarios. The predicted reliability of SeqWizard was consistent with the reliability report of SeqWizard in its beta testing stage. This indicated the suitability of the reliability BBN constructed in this thesis and the effectiveness of the BBN model. In addition to reliability, the number of faults introduced to SeqWizard, the number of faults found during testing, and the number of latent faults remained in SeqWizard were also obtained from the reliability BBN. These predicted results can be used by decision-makers in software development. The experience obtained in this analysis can be extended to other applications of the BBN model. We believe that the BBN model is not just effective for software reliability prediction and estimation, it is also effective for various decision-making support in software engineering.

5.2 Future Work

There are more than 40 software reliability models in the open literature. Six of them, Musa's models, the Jelinski-Moranda model, the Geol-Okumoto NHPP model, Shooman's model, the Littlewood-Verral model, and the Bayesian Belief Networks (BBN) model, were investigated in this thesis. The BBN model has been analyzed in detail through applying it to predict the reliability of the SeqWizard software. Some other models need to be investigated and analyzed in detail in order to explore their effectiveness for software reliability prediction and/or estimation.

REFERENCES

[Bouissou et al. 99] M. Bouissou, F. Martin, and A. Ourghanlian, "Assessment of a Safety-Critical System Including Software: A Bayesian Belief Network for Evidence Sources", *Proceedings of the Annual Reliability and Maintenance Symposium*, pp. 142-150, Washington D. C., January 1999.

[Cowell et al. 99] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, and D. J. Spiegelhalter, *Probabilistic Networks and Expert Systems*, Springer-Verlag, New York, NY, 1999.

[Dagum et al. 95] P. Dagum, A. Galper, E. Horvitz, and A. Seiver, "Uncertain Reasoning and Forecasting", *International Journal of Forecasting*, Vol. 11, No. 1, pp. 73-87, March 1995.

[Dhillon 87] B. S. Dhillon, *Reliability in Computer System Design*, Ablex Publishing Corporation, Norwood, NJ, 1987.

[Fenton and Neil 99] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models", *IEEE Transactions on Software Engineering*, Vol. 25, No. 5, pp. 675 – 689, September-October 1999.

[Hansen et al. 99] C. K. Hansen, K. P. LaSala, S. Keene, and A. Coppola. "The Status of Reliability Engineering Technology", *The Journal of the Reliability Analysis Center (RAC)*, Vol. 5, No. 2, pp. 332-339, June 1999.

[Hugin 90] Hugin Expert A/S, creation date: 1990, access date: May 2000, available at: http://www.hugin.com/.

[Lauritzen and Spiegelhalter 88] S. L. Lauritzen and D. J. Spiegelhalter, "Local Computation with Probabilities on Graphical Structures and Their Application to Expert Systems (with discussion)", *Journal of the Royal Statistical Society*. Series B, Vol. 50, No. 2, pp. 157-224, April 1988.

[Liang and Trivedi 99] Y. Liang and K. S. Trivedi, "Confidence Interval Estimation of NHPP-Based Software Reliability Models", *Proceedings of 10th International Symposium on Software Reliability Engineering*, pp. 6-11, Boca Raton, Florida, November 1999.

[Lyu and Nikora 92] M. R. Lyu and A. Nikora, "Applying Reliability Models More Effectively (Software)", *IEEE Software*, Vol. 9, No. 4, pp. 43-52, July 1992.

[Lyu 96] M. R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Musa et al. 87] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability*. McGraw-Hill Publishing Company, New York, NY, 1987.

[Musa et al. 90] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Publishing Company, New York, NY, 1990.

[Neil and Fenton 96] M. Neil and N. E. Fenton, "Predicting Software Quality using Bayesian Belief Networks", *Proceedings of the 21st Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Seattle, Washington, December 1996.

[Neil et al. 99] M. Neil, N. E. Fenton, and L. Nielsen, "Building Large-Scale Bayesian Networks", *Knowledge Engineering Review*, Vol. 3, No. 5, pp. 1-31, July 1999.

[Neufelder 93] A. M. Neufelder, *Ensuring Software Reliability*. Marcel Dekker Inc., New York, NY, 1993.

[Norsys 95] Norsys Software Corporation, creation date: 1995, access date: June 2000, available at: http://www.norsys.com/.

[Pearl 97] J. Pearl, "Graphical Models for Probabilistic and Causal Reasoning", *Computer Science and Engineering Handbook*, pp. 697-714, CRC Press Inc., Boca Raton, Florida, 1997.

[Pham 95] H. Pham, *Software Reliability and Testing*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[Schach 97] S. R. Schach, *Software Engineering with JAVA*, Times Mirror Higher Education Group, Chicago, IL, 1997.

[Samadzadeh and Edwards 88] M. H. Samadzadeh and W. R. Edwards, "A Classification Model of Software Comprehension", *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, p. 541, Kailua Kona, Hawaii, January 1988.

[SERENE 90] Hugin Expert A/S SERENE Demo, creation date: 1990, access date: May 2000, available at: http://www.hugin.com/serene/.

[Sommerville 97] I. Sommerville, *Software Engineering*, 5th edition, Addison-Wesley, Harlow, England, 1997.

[Yu et al. 99] D. C. Yu, T. C. Nguyen, and P. Haddawy, "Bayesian Network Model for Reliability Assessment of Power Systems", *IEEE Transactions on Power Systems*, Vol. 14, No. 2, pp. 426-432, May 1999.

[Zand and Samadzadeh 94] M. K. Zand and M. H. Samadzadeh, "Software Reuse: Issues and Perspectives", *IEEE Potentials*, Vol. 13, No. 3, pp. 15-19, August-September 1994.

[Ziv and Richardson 97] H. Ziv and D. J. Richardson, "Constructing Bayesian-Network Models for Software Testing and Maintenance Uncertainties", *Proceedings of International Conference on Software Maintenance*, pp. 100-109, Bari, Italy, October 1997.

GLOSSARY

AIAA – American Institute of Aeronautics and Astronautics. The nonprofit AIAA is the principal society and voice serving the aerospace profession. Its primary purpose is to advance the arts, sciences, and technology of aeronautics and astronautics and to foster and promote the professionalism of those engaged in these pursuits.

Bellcore – Bell communications and research.

BBN – Bayesian Belief Networks.

CASE -- Computer-Aided Software Engineering. CASE is the use of computer-based support in software development process.

CASRE – Computer-Aided Software Reliability Engineering. It is an automated software reliability tool developed by NASA in 1994.

Conditional Probability Table (CPT) – In a Bayesian Belief Network, each node contains the states of the random variable it represents and a conditional probability table (cpt). The cpt of a node contains probabilities of the node being in a specific state given the states of its parents.

DAG – Directed Acyclic Graph.

DACS – Data & Analysis Center for Software.

DNA – Deoxyribose Nucleic Acid. A nucleic acid that carries the genetic information in the cellular organisms and is capable of self-replication and synthesis of RNA (Ribose Nucleic Acid). DNA consists of two long chains of nucleotides twisted into a double helix and joined by hydrogen bonds between the complementary bases adenine(A) and thiamine(T) or cytosine(C) and guanine(G). The sequence of nucleotides determines individual hereditary characteristics. A DNA sequence can be considered as a string of characters A, T, C, and G.

ESTM – Economic Stop Testing Model.

Execution Time – The CPU time that is actually spent by a computer in executing software.

Failure Intensity Function – This function represents the rate of change of the cumulative failure.

Failure Rate – The probability that a failure per unit time occurs in the interval, say, $[t, t+\Delta t]$, given that a failure has not occurred before time t. That is,

$$\begin{aligned}\text{Failure Rate} &= P(t <= T < t+\Delta t \mid T>t) / \Delta t \\ &= P(t <= T < t+\Delta t) / (\Delta t \times P(T>t)) \\ &= (F(t+\Delta t) - F(t)) / (\Delta t \times R(t))\end{aligned}$$

where $F(t)$ is the probability of failure by time t, and $R(t)$ is the reliability function defined in Chapter III of this report.

GUI – Graphical User Interface.

Hazard Rate – Defined as the limit of the failure rate as the interval size approaches zero, that is, $\Delta t \rightarrow 0$, $z(t) = \lim(F(t+\Delta t) - F(t)) / (\Delta t \times R(t))$
where $F(t)$ is the probability of failure by time t, and $R(t)$ is the reliability function defined in Chapter III of this report.

Idiom – The basic building block of the Bayesian Belief Networks developed by Fenton and Neil based on generic uncertain reasoning.

KLOC – Thousand Lines Of Code.

LAAS – Laboratory of Analysis and Architecture of System.

LOC – Lines Of Code.

Mean Time To Failure (MTTF) – The expected time that the next failure will be observed. Also known as Mean Time Between Failures (MTBF).

MUNIN – MUscle and Nerve Inference Network. A causal probabilistic network constructed to diagnose muscle and nerve diseases through analysis of bioelectrical signals from muscle and nerve tissues.

NHPP – NonHomogeneous Poisson Process.

NSWC – Naval Surface Warfare Center.

Operational Profile – A set of operations and the frequency of each operation specified for software.

RSC Ltd. – Reliability and Statistical Consultants, Ltd.

Software Life Cycle – Software development generally includes eight phases: requirement establishing, specification analysis, planning, design, implementation, integration and system testing, maintenance, and retirement. These phases are collectively called software life cycle.

Software Reliability – The probability of failure-free software operation for a specified period of time in a specified environment.

Software Reliability Engineering – The application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems.

Software Reliability Model – A software reliability model specifies the general form of dependency of the failure process on the principal factors that affect it. Such factors include: fault introduction, fault removal, and the operational environment.

SMERFS – Statistical Modeling and Estimation of Reliability Functions of Software. SMERFS is an automated software reliability tool developed by Naval Surface Warfare Center in 1983.

SRMP – Software Reliability Modeling Programs. SRMP is an automated software reliability tool developed by the Reliability and Statistical Consultants, Ltd. (England) in 1988.

2

VITA

Qiaolan Wan

Candidate of the Degree of

Master of Science

Thesis: SOFTWARE RELIABILITY MODELS: A SURVEY AND AN ANALYSIS

Major Field: Computer Science

Biographical:

Personal Data: Born in Gansu, P. R. China, on October 4, 1974, daughter of Mr. Tingdi Wan and Ms. Guanglian Luo, and wife of Bingdong Li.

Education: Received Bachelor of Science in Biology from Northwest Agricultural University, Xi'an, P. R. China in July 1993; received Master of Science in Biology from The Graduate School of Chinese Academy of Agricultural Science, Beijing, P. R. China in July 1995; completed the requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in July, 2000.

Experience: Employed by Oklahoma State University, Molecular Genetics and Microbiology Science Department as a Software Developer, September 1998 to July 2000.