AVAILABILITY AND RELIABILITY ANALYSIS
OF COMPUTER SOFTWARE SYSTEMS
CONSIDERING
MAINTENANCE AND SECURITY ISSUES

XIONG CHENG-JIE

NATIONAL UNIVERSITY OF SINGAPORE
2011
AVAILABILITY AND RELIABILITY ANALYSIS OF COMPUTER SOFTWARE SYSTEMS CONSIDERING MAINTENANCE AND SECURITY ISSUES

XIONG CHENG-JIE
(B. Eng), WUHAN UNIVERSITY

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF INDUSTRIAL AND SYSTEMS ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE
2011
First and foremost my greatest gratitude goes to my main supervisor, Prof. Xie Min, who supported me both in research and in practice all through my PhD candidature and whose wisdom is so charming that enlightened my research path. He is not only a well-respected advisor in research, but also more a spiritual tutor as well as a great friend. His knowledge and patience helped me transform from a beginner to a junior research without anxiety or frustration, which I am extremely grateful for. What is more, his noble personality and passion for research always spurs me to search more for research and life. I would have achieved much less if I had not been under his supervision.

I am also heartily thankful to my co-supervisor, A/Prof. Ng Szu Hui, for her guidance and very detailed suggestions in my research, especially when I was undergoing the manuscript reviewing process. Her serious attitude towards research always reminds me to get rid of fickleness and utilitarianism, which was fatal in research. Her positive life style and friendliness also made me feel more inspired when I was her teaching assistant. I also gained lots of teaching experience from her. I am really grateful for her directions.

Then I also want to express my gratefulness to the Department of Industrial and Systems Engineering and the National University of Singapore, for providing me scholarship and an
excellent environment to study and do research. I would have missed the most precious life experience if I had not won a chance to study in Singapore.

I would like to thank A/Prof. Poh Kim Leng and Dr. NG Tsan Sheng, Adam, for serving as my oral qualification exam committee and provided me lots of deep insights for my further research. I also greatly thank Ms. Ow Lai Chun and Mr. Lau Pak Kai, for their excellent administrative and technical support to my PhD study. The warmest gratefulness also goes to A/Prof. Tang Loon Ching, who interviewed and encouraged me to join the Department of Industrial and System Engineering in Shanghai 5 years ago. I would not be able to finish this thesis, or even to start writing this thesis, without their help.

As a member of the Quality and Reliability Lab, I also benefited a lot from my lab mates. With all of them, I had a great post-graduate life for 4 years. Special thanks goes to Li Yanfu and Shen Yan, who helped me a lot in life and at the same time in research, especially in research proposal and thesis writing. I would also like to thank Qian Yanjun, Yin Jun and Zhou Peng, for their help and advices that made my life and research easier.

Finally, I would like to express my deepest thank to my parents, for their continuous encouragements and support for more than 20 years. Furthermore, to Mao Chengting, thank you for your understanding and encouragements.
# Table of Contents

Acknowledgement ................................................................................................................ I

Summary ................................................................................................................................ IV

List of Tables .......................................................................................................................... VII

List of Figures ........................................................................................................................ VIII

List of Notations and Acronyms ............................................................................................ IX

Chapter 1 Introduction ........................................................................................................... 1

  1.1 Focus of this thesis ........................................................................................................... 2
  1.2 Introduction to Quality Metrics of Computer Software Systems ....................................... 2
    1.2.1 Software Reliability ................................................................................................... 3
    1.2.2 Software Availability ................................................................................................. 3
    1.2.3 Software Service Reliability .................................................................................... 4
  1.3 External Factors Affecting Software Quality ................................................................... 5
    1.3.1 Software Maintenance ............................................................................................... 5
    1.3.2 Malicious Software .................................................................................................. 6
  1.4 Research Motivation ..................................................................................................... 7
  1.5 Thesis Organization ....................................................................................................... 8

Chapter 2 Literature Review ................................................................................................... 10

  2.1 Reviews on Software Reliability Modelling .................................................................. 10
    2.1.1 Basic Software Reliability Models .......................................................................... 10
    2.1.2 Software Reliability Model Extensions ...................................................................... 22
  2.2 Reviews on Software Availability Related Problems ....................................................... 25
    2.2.1 Software Maintenance and Software Availability ...................................................... 26
    2.2.2 Software Availability Assessment .......................................................................... 32
    2.2.3 Malicious Software and Its Impact on Software Quality ............................................ 36

Chapter 3 Software Availability Modelling and Application Extensions ............................... 40

  3.1 Introduction .................................................................................................................... 40
3.2 Software Maintenance Activities and Software Availability ........................................ 43
  3.2.1 Decomposition of Software Maintenance Activities ........................................ 43
  3.2.2 Software Maintenance Efforts and Software Maintenance Modeling .................. 46
  3.2.3 Software Maintenance Modeling—A Numerical Example .................................. 51
  3.2.4 Validation of Software Maintenance Modelling ............................................... 57
  3.2.5 Software Availability Modeling Considering Software Maintenance .................... 58

3.3 Software Maintenance Policy Considering Software Availability Constraints .......... 67
  3.3.1 Re-investigation on Software Maintenance and Software Maintenance Policy ........ 67
  3.3.2 Software Maintenance Policy Modeling with Consideration of Software Availability and Cost Constraints ............................................................. 71
  3.3.3 Validation of Policy Optimality—A Numerical Example ................................... 75

3.4 Summary and Conclusion Remarks ........................................................................ 80

Chapter 4  Software Availability Assessment of N-version Programming Systems ....... 82

  4.1 Introduction ............................................................................................................. 82
  4.2 Software Availability Modeling of N-version Programming Systems .................... 85
    4.2.1 Structure Decomposition of N-version Programming Systems ....................... 86
    4.2.2 Proposed Model of Software Availability of N-version Programming Systems ... 89
  4.3 Impact of N-version Programming on Software Availability .............................. 98
    4.3.1 Failure and restoration rates for different versions of software ......................... 98
    4.3.2 Impact of N-version Programming on Software Availability: A Simulation Approach .... 101
    4.3.3 Optimal Software Structure under Software Availability and Budget Constraints .... 112

Chapter 5  Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack ............................................................................................................ 119

  5.1 Introduction ........................................................................................................... 120
  5.2 Distributed Software Systems and Malware Epidemics: The Homogeneous Scenario ... 122
    5.2.1 Modelling of Distributed System under Malware Attack: The Homogeneous Scenario .... 123
    5.2.2 Derivation of Software Service Reliability and Software Availability .................. 129
    5.2.3 Numerical Examples ....................................................................................... 131
  5.3 Distributed Software Systems and Malware Epidemics – A Revisit ....................... 134
    5.3.1 A Continuum State Reliability Model of Individual Nodes ............................. 135
    5.3.2 Virus Epidemic Model: The General Scenario .............................................. 138
5.3.3 An Illustrative Example ............................................................................................................. 140
5.4 Service Reliability Modeling of Distributed System Considering Malware Epidemics ........... 145
  5.4.1 Service Reliability Model without Communication Channel Failure ............................... 146
  5.4.2 Service Reliability Model with Communication Channel Failure ..................................... 149
  5.4.3 Computation of Service Reliability ....................................................................................... 151
  5.4.4 A Numerical Example ........................................................................................................... 153

Chapter 6 Conclusions and Future Works ....................................................................................... 160
  6.1 Conclusions and Contributions ............................................................................................... 160
  6.2 Directions of Future Research ................................................................................................. 162

Bibliography ...................................................................................................................................... 165

Appendix Service Reliability Calculation Algorithm .......................................................................... 179
Summary

Reliability and availability have long been considered as critical metrics of high quality software systems. However, as plenty of research efforts have been devoted into the field of software reliability, only a little has been documented in the aspect of software availability. In the mean time, traditional ways of analyzing software quality often ignores the impact of environmental factors, such as software maintenance and software security issues. Realizing the importance of software availability to software quality study, this thesis first focuses on the modeling of software availability and some application extensions (Chapter 3). Then the model is further extended to analyze the availability issues of fault-tolerant software systems (Chapter 4), followed by a quality analysis of distributed software system considering malicious software attack (Chapter 5).

The primal focus of this thesis is to develop a proper model to assess availability of software systems by analyzing feedback data (Chapter 3). To achieve this purpose, the origination of software availability problems is first analyzed. We assert that software maintenance is solely responsible for causing software availability problems and a rate-based model for describing software maintenance process is proposed. Based on the maintenance model, we incorporate the existing NHPP software failure models and propose a general approach for systematically calculating software availability. In order to check the effectiveness
of our proposed models, we apply our model to a real-life industrial case to show its ability in helping seeking the optimal software maintenance policies.

Besides the general modeling for calculating software availability, we also investigated the availability problems of fault-tolerant software systems (Chapter 4). The N-version programming technique is covered in this thesis and we proposed a Markov chain based software availability model to assess the availability of this special kind of software system. Interactions among different versions are explicitly considered and analyzed. Numerical analysis clearly reveals the positive impact of the N-version technique on enhancing software availability and we also propose a method for determining the optimal software structure from the availability cost-effectiveness point of view.

Moreover, as the threat of malicious software increases day by day with the quick popularization of internet, the analysis of the relationship between software quality degradation and malicious software attack forms another important part of this thesis. Hence, Chapter 5 analyzed the problem of quality degradation of distributed software systems by considering malicious software attack. We first analyze the spreading path of malicious software within distributed software systems via the Markovian approach and derive the availability and reliability metrics of the system. Due to the difficulty in mathematical tractability of Markovian models, we revisit the malicious software epidemic problem using the continuum state model, which simplifies the mathematical calculation and provides us a highly abstract view of the whole distributed system. A general model of virus epidemic within distributed software systems is proposed based on the continuum state model, based on which software quality metrics is easy
to obtain. Furthermore, we derive and propose an algorithm for computing the service reliability, which is easy for computer utilization.
List of Tables

Table 3-1 Maintenance Data of Apache Release 2.0.35 ................................................................. 54
Table 3-2 Summary on Model Estimates and Goodness-of-fit for Apache Release 2.0.35 ....... 54
Table 3-3 Field Data with Fitted Data for Apache 2.0.35 ................................................................. 56
Table 3-4 Simulation Data for Software Failure & Maintenance ...................................................... 63
Table 3-5 MLE Estimation Results ..................................................................................................... 64
Table 3-6 Availability Prediction ........................................................................................................ 66
Table 3-7 Simulation Parameters Configuration .................................................................................. 77
Table 3-8 Simulation Results ................................................................................................................ 78
Table 3-9 Sensitivity Analysis ............................................................................................................. 79
Table 3-10 Interaction Analysis ........................................................................................................... 80
Table 4-1 Details of 3 Versions in Scenario One .................................................................................. 102
Table 4-2 Average Availability of Different Software Systems in Scenario One ....................... 104
Table 4-3 Details of 3 versions in Scenario Two .................................................................................. 106
Table 4-4 Average Availability of Different Software Systems in Scenario Two ..................... 107
Table 4-5 Details of 3 versions in Scenario three .............................................................................. 109
Table 4-6 Average availability of different software systems in Scenario three ....................... 110
Table 4-7 Maintenance Cost Factors ................................................................................................ 117
Table 4-8 Optimal Software Structure and Maintenance Policy ...................................................... 117

Table 5-1 Infection, Restoration and Failure Probability ................................................................. 132
Table 5-2 Weight of Edges and Virus Defense Parameter ............................................................... 140
Table 5-3 Set of Nodes ....................................................................................................................... 155
Table 5-4 Failure Probability of Edges .............................................................................................. 155
Table 5-5 Processing Speed of Nodes ............................................................................................... 155
List of Figures

Figure 1-1 Thesis organization ........................................................................................................................... 8
Figure 2-1 Software Reliability Models Classification .......................................................................................... 13
Figure 3-1 Testing and debugging during software testing phase ............................................................................. 44
Figure 3-2 A demonstration of software operation and maintenance during the operational phase ......................... 46
Figure 3-3 Actual versus Fitted Number of Corrected Faults with MLE and LSE for Apache 2.0.35 .......................................................... 57
Figure 3-4 Plot of Simulated Software Process ........................................................................................................ 64
Figure 3-5 Actual Data versus Estimated Curve ........................................................................................................ 65
Figure 3-6 Demonstration of Operation and Maintenance of Software Systems ......................................................... 70
Figure 4-1 Traditional Software vs. N-version Programming Software ................................................................. 88
Figure 4-2 State Transition Diagram of Sub-version \(i\) ............................................................................................. 91
Figure 4-3 Scenario One: Instant Availability ........................................................................................................ 104
Figure 4-4 Scenario One: Average Availability ....................................................................................................... 105
Figure 4-5 Scenario 2: Instant Availability ........................................................................................................... 107
Figure 4-6 Scenario Two: Average Availability ..................................................................................................... 108
Figure 4-7 Scenario Three: Instant Availability .................................................................................................... 110
Figure 4-8 Scenario Three: Average Availability .................................................................................................. 111
Figure 5-1 Structure of a N-nodes homogeneous distributed system ........................................................................ 124
Figure 5-2 Transition graph of state \(\{H,I,D\}\) ......................................................................................................... 127
Figure 5-3 Software Service Reliability and Software Availability Plot over \(t\) ............................................................ 133
Figure 5-4 A Distributed System with 5 Nodes ....................................................................................................... 141
Figure 5-5 Plot of \(\mu_i(t)\) over Time ......................................................................................................................... 144
Figure 5-6 A distributed system with 5 nodes ........................................................................................................ 156
Figure 5-7 Service Reliability of the 5-node Distributed System over Time ........................................................... 158
# List of Notations and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NHPP</td>
<td>Non-Homogeneous Poisson Process</td>
</tr>
<tr>
<td>MLE</td>
<td>Maximum Likelihood Estimation</td>
</tr>
<tr>
<td>LSE</td>
<td>Least Square Estimation</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time to Failure</td>
</tr>
<tr>
<td>MTTR</td>
<td>Mean Time to Repair</td>
</tr>
<tr>
<td>$L$</td>
<td>The Maximum Likelihood Function</td>
</tr>
<tr>
<td>${X(t)}$</td>
<td>A stochastic process</td>
</tr>
<tr>
<td>$R(t)$</td>
<td>Software Reliability at time $t$</td>
</tr>
<tr>
<td>$A(t)$</td>
<td>Software Availability at time $t$</td>
</tr>
<tr>
<td>$a(t)$</td>
<td>The probability that software is functioning at time $t$</td>
</tr>
<tr>
<td>$\lambda, \mu$</td>
<td>Failure intensity/Transition rate</td>
</tr>
<tr>
<td>$\lambda(n, t)$</td>
<td>Software failure rate</td>
</tr>
<tr>
<td>$\mu(n, t)$</td>
<td>Software Repair rate</td>
</tr>
<tr>
<td>$Pr$</td>
<td>Probability</td>
</tr>
<tr>
<td>$m(t)$</td>
<td>Mean value function</td>
</tr>
<tr>
<td>$m_o(t)$</td>
<td>Mean value function of the operation process</td>
</tr>
<tr>
<td>$m_r(t)$</td>
<td>Mean value function of the maintenance process</td>
</tr>
<tr>
<td>$w(t)$</td>
<td>Testing Effort rate function</td>
</tr>
<tr>
<td>$h(t)$</td>
<td>Total number of software maintenance participants at time $t$</td>
</tr>
<tr>
<td>$W(t)$</td>
<td>Cumulative Testing Effort function</td>
</tr>
<tr>
<td>$U(t)$</td>
<td>Arithmetic sum of subsequent time to failure</td>
</tr>
<tr>
<td>$D(t)$</td>
<td>Arithmetic sum of subsequent time to repair</td>
</tr>
<tr>
<td>$n$</td>
<td>Total number of sub versions in the N-version programming system</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Number of initial faults in sub version $i$</td>
</tr>
</tbody>
</table>
\( \lambda_{i,j} \)  
Failure rate of sub version \( i \) when \( j \) failures have been encountered

\( \mu_{i,j} \)  
Restoration rate of sub version \( i \) when \( j \) failures have been encountered

\( p^j_{i,(0,k,0)} \)  
Probability that version \( i \) will be in working state at time \( t \) after \( k \) faults are to be removed, given that the software is currently in the working state and \( j \) faults have been removed previously

\( (j(t),w) \)  
Pair that represents a sub version state

\( A^i(t) \)  
Instant software availability of sub version \( i \) at time \( t \)

\( A(t) \)  
Instant software availability at time \( t \)

\( \bar{A}^i(t) \)  
Average software availability over the period \((0,t]\)

\( D_i \)  
Initial hazard rate of sub version \( i \)

\( E_i \)  
Initial failure restoration rate of sub version \( i \)

\( C_{total} \)  
Total cost during the software operational cycle

\( C_o \)  
Cost of software operation

\( C_m \)  
Cost of software maintenance

\( C_r \)  
Risk cost if the software system is unavailable

\( c_o \)  
Expected unit time cost of software operation

\( c_i \)  
Expected unit time cost of maintenance cost of sub version \( i \)

\( c_r \)  
Expected unit time cost if the software system is unavailable

\( L \)  
Expected length of software operation cycle

\( A_{req} \)  
Requirement of software availability

\( \lambda_i \)  
Failure rate of computer nodes

\( \mu_r \)  
Restoration rate

\( \lambda_m \)  
Malware infection rate

\( \lambda_a \)  
Attacking rate of a malware-infected node

\( p_{(H,I,D)}(t) \)  
Probability that the system is in state \( (H,I,D) \) at time \( t \)
\( \alpha \) Service rate of each computer node of an exponential distribution

\( H \) The number of normal computer nodes

\( I \) The number of computers that are infected by malware

\( D \) The number of computers that fail

\( N \) The total number of computers: \( N = H + I + D \).

\( T_E \) Required finishing time of a single service request

\( \Omega \) Range of continuous state \( \Omega = [0,1] \), where 0 indicates the perfect functioning state and 1 is the complete failure state.

\( N \) Total number of nodes in the computer network

\( G \) The undirected graph representing the computer network

\( V \) The set of nodes in the computer network

\( v_i \) Node \( i \) in the computer network

\( L \) The set of communication channels in the computer network

\( l_{ij} \) The communication channel that links node \( v_i \) and \( v_j \)

\( x_i(t) \) The state of node \( i \) at time \( t \), \( x \in \Omega \)

\( F(x_i,t) \) The distribution function of state \( x_i \)

\( \mu_i(t) \) The expectation of the state \( x_i(t) \)

\( \Phi_i \) The neighborhood set of node \( v_i \)

\( \delta_i \) Defense parameter at node \( v_i \)

\( D \) The total amount of data to be transmitted in the distributed system

\( \theta_i \) Data processing speed of node \( v_i \)
Chapter 1 Introduction

Although computer software systems are relatively young in industry when compared to other hardware systems, there is a dramatic increase in demand for software in the market, software systems are playing more and more important roles day by day. Software becomes more and more complex and difficult to design, produce and maintain, which limits the development of software technology. In the mean while, with the fast development of hardware, which complies with the “Moore’s Law”, people are continuously upgrading their requirements on software systems, starting from “functioning efficiently” (Jelinski and Moranda, 1972) in the early days when computational capacity was the main barrier to “functioning elegantly” (Pham, 2003) to nowadays when the quality of a software system is of main concern. The quality of a software system is vital to the success of a software system. Software quality analysis is a very active research field as it was more than 50 years ago, when the first generation of software systems came into being (Brooks, 1975; Schich and Wolverton, 1973). Although the quality of software systems consists of many different aspects, starting from coding patterns to user interface interactions, Reliability and Availability are by far the most important metrics for estimating quality of software systems, which was discussed more than thirty years ago in the de-facto software engineering bible – “The Mythical Man Month” (Brooks, 1975).
1.1 Focus of this thesis

The objective of this thesis is to propose a proper framework for quantitatively estimating software quality, under which the problem of software maintenance, software structure and software security issues can be analyzed in an integrated manner.

1.2 Introduction to Quality Metrics of Computer Software Systems

Quality is vital for all kinds of projects. The word “quality” is somehow a qualitative term. However, in software industry, quality needs to be assessed quantitatively by different metrics. There are many standard methods and frameworks for quality assurance and quality control (Kan, 2003) of projects, but such mechanisms can hardly be directly applied to software projects. Unlike the manufacturing processes, software systems are sets of logics, which do not degrade or deviate and there are only design and coding flaws, rather than manufacturing flaws. Given the same input, a software system (not including special software systems such as concurrent programs, which are not concerned in this thesis) will always generate the same output at any time, no matter right or wrong (Xie, 1991; Lyu, 1996; Pham, 2003). As such, it is hard to define the quality of a software system in a single term and additional metrics are needed to help assess the quality of a software system. Research in this area started from the technical point of view of software practitioners in software reliability (Jelinski and Moranda, 1972; Goel and Okumoto, 1979; Shanthikumar, 1981) and now has expanded to the consumers’ point of view of software end users in software availability (Tokuno and Yamada, 2003) and software service reliability (Dai et al, 2003; Dai and Levitin, 2007).
1.2.1 Software Reliability

There are many definitions of software reliability, but most authors consider that software reliability represents the probability of failure-free software operation for a specified period of time in a specified environment (Xie, 1991). Software reliability is widely regarded as the most important quality metrics and often used as an indicator for software release policy (Xie and Yang, 2003). Researchers began to analyze software reliability during the late 1960s and it was first analyzed as a standard probability problem using mathematical approaches (Littlewood and Verrall, 1973). With further understanding about the nature of software failure process, people began to switch to study this problem from the stochastic point of view (Jelinski and Moranda, 1972; Geol and Okumoto, 1979) and fruitful results were derived (Pham, 2003). Although software reliability has been a hot research topic for more than thirty years, this field is still attractive and new discoveries and research results are reported every year.

1.2.2 Software Availability

The problem of software availability was first reported in the 1970s (Trivedi and Shooman, 1975), but it did not receive wide recognition because at that time the computational constraints of computer hardware urged people to seek for efficiency rather than elegance. However, with the development of hardware technology and an increase population who are enjoying services provided by software, software availability related problems cannot be ignored (Tokuno and Yamada, 2003).
When compared to software reliability, the literature in software availability is not vast and there is no standard definition of software availability. Some authors regard software availability as the probability of software being in a working state at a given time in a given environment (Kim et al, 1982; Okumoto and Geol, 1979; Tokuno and Yamada, 2003) while others calculate software availability as the percentage of total scheduled service time when systems are operational and ready to provide service (Gokhale and Trivedi, 1999; Zhang and Pham, 2002). Although the two definitions sound somehow different, it can be shown that the two can be unified (Please refer to Chapter 3).

Software availability is also vital to the quality of software systems. Software availability is important because software systems that require high availability are usually serving a large number of people and even a brief outage may cause a huge loss. According to the records of Kan (2003), if the software system in the two Unisys Corporation mainframe computer systems at the New York Clearing House is unavailable for one second, there will be losses of 14 million US dollars. Although the topic of software availability is covered in literature (Tokuno and Yamada, 2003, Gokhale and Trevidi, 1999; Zhang and Pham, 2002; Gokhale et al, 2004), it has not yet been systematically studied and has been treated as a by-product of software reliability analysis.

1.2.3 Software Service Reliability

Software service reliability is relatively a new term when compared to software reliability and availability, but it successfully reflects people’s requirements of a modern software system. Software service reliability represents the probability of software system’s successful completion.
of a specific task within a specific time period in a specific environment (Dai et al., 2003). As the definition indicates, software service reliability is determined by two criteria, namely correctness and timeliness. This metric was proposed because together with software reliability and software availability, they represent three most basic requirements when people are using software systems – software needs to be correct (Reliability); software need to be accessible (Availability) and software needs to be stable (Service Reliability).

1.3 External Factors Affecting Software Quality

For hardware systems, the quality is not only determined by their internal properties. There are problems like burn-in phase and wear-out phase, and external factors such as temperature, humidity and altitude can all possibly affect their performance. For software systems, although they do not degrade with time, their performance can be affected by external factors. For example, performance of video conference software can be greatly lowered down due to slow network connection. Among most external factors, software maintenance and malicious software are of the most important factors that affect software quality.

1.3.1 Software Maintenance

Software maintenance is carried out during the operational phase and usually hampers normal software operation. The software system usually cannot be online until software maintenance is done. Two major characteristic of software maintenance is that it takes both time and efforts. Many authors have been devoting their efforts to find better solutions of software
maintenance so that fewer efforts are consumed (Ahn et al., 2003; April et al., 2005; Ahmed, 2006) while few have ever covered the fact that the time in maintenance also affects software quality. In fact, long software maintenance can lower software availability and correction in source code can affect both software reliability and software service reliability. What is more, for certain types of software such as open-source software, maintenance is so highly integrated with development that it is impossible to ignore the impact of software maintenance.

1.3.2 Malicious Software

The problem of software security has never been so severe, thanks to the prevailing Internet. Malicious software, which is often referred to as malware, can be found in many different forms such as software virus, Trojan-horses and so on. They consume resources and reproduce themselves whenever possible, resulting in degrading the whole computer system and making software systems hard to respond to normal service requests. It is normal to see a healthy software system turn into completely deaf and mute when it is infected by viruses. However, as one major and as well as the most deadly threat to software quality, malicious software is not thoroughly studied (Kondakci, 2008; Kondakci, 2009) and only limited documentation can be found. In most cases, it is almost certain that malware can degrade the performance and quality of software systems and it is worth conducting a thorough analysis of malware’s impact on software quality.
1.4 Research Motivation

Quality metrics (reliability/availability/service reliability) modeling play an essential role in the analysis of software quality. Many researchers focused on proposing new models that can better fit historical data, and these models were used to generate quantitative results. Some of these results could be used for optimization purpose (Huang and Lyu, 2005), whilst others are just figures for reference since the models were not properly integrated with practice. What is more, as shown in the previous two sections, we assert that the quality analysis of software systems cannot exclude external factors and many of the existing models do not provide satisfactory results in some occasions. To overcome these drawbacks, it is necessary to take software maintenance, software structure and software security problems into the consideration of software quality metrics modeling.

As discussed above, many researchers have focused on three factors. Software maintenance has never escaped from the focus of researchers due to its high cost of capital and labor and countless efforts are devoted into this field in the search for optimal maintenance policies (Huang and Lyu, 2005; Xie and Yang, 2003; Dai and Levitin, 2007). Many researchers have also covered the problem of software reliability over different software structures (Dai et al., 2004) and software security problems have always been a high end topic in both industry and academia with tens of thousands of software practitioners participating (Kondakci, 2009). However, most of the existing work treats these factors as isolated problems and lacks an integrated image of how these factors affects the quality of software system as a whole. In my opinion, there is still a lack of research work to investigate and integrate these problems under a unified framework and we are motivated to conduct such an analysis.
1.5 Thesis Organization

A number of journal/conference papers have been published under this objective. The research works are grouped into three categories: works on software availability modeling and applications (Chapter 3); works on software availability analysis over fault-tolerant software structures (Chapter 4); works on software availability and reliability analysis over software security issues (Chapter 5). The rest chapters of this thesis, namely Chapter 2 and 6 are literature review and conclusions for future research, respectively. Figure 1-1 shows the work presented in this thesis and their internal relationship.

![Figure 1-1 Thesis organization](image)

The entire thesis is written based on the motivation of quantitatively analyzing software quality in an integrated manner and is of both significant practical and theoretical value. In
practice, we believe this work can help software enterprise to make rational decisions both in the development and deployment phase when facing complicated problems with multiple constraints. Theoretically speaking, the studies in this thesis extended existing research in software availability, reliability and service reliability modeling with integration of external factors and established a novel approach for modeling malware epidemics within computer networks.

In order to present a better understanding of each research work, the background information is presented in the literature review in the next chapter.
Chapter 2       Literature Review

Software quality receives great attention of many software practitioners and various kinds of methods have been proposed to estimate software quality. Other issues that directly affect software quality, such as software maintenance, software structure and software security issues have also been covered by many authors. This chapter provides a detailed summary of the literature that covered the above mentioned topics which are published mainly in the past decade and are related to the foundations of this thesis. Software reliability models and extensions serve as the basis of software quality research and this thesis is also motivated to extend the existing software quality research. Thus, software reliability models will be reviewed first. Then we focus on the problems of software availability. The origination of software availability problem – software maintenance is covered first and then we will go through the existing models that talk about software availability. Finally, side issues such as software security will also be covered.

2.1 Reviews on Software Reliability Modelling

2.1.1 Basic Software Reliability Models

Reliability is the first software quality metric that people pay attention to (Brooks, 1975). Software reliability is commonly recognized as the probability of failure-free operation of a software system in a specific operation environment for a given period of time (Xie, 1991; Lyu, 1996; Pham, 2003). There are plenty of Software Reliability Models (SRM) ever since the late
1960s; however, the modelling of software failures has always been the main concern of modelling software reliability.

Existing literature on software reliability is vast and intensive with many authors having proposed many different SRMs using different assumptions and approaches. Some review and classification work of SRMs can be found in Ramamoorthy and Bastani (1982), Goel (1985) and Xie (1991). The classification scheme of Xie (1991) is the most popular and most widely adopted one and we also follow and extend the classification scheme of Xie (1991) in this thesis.


Among the above mentioned model categories, statistical data analysis methods, input-domain-based models, seeding and tagging models and software metrics models focus on the overall prediction of software failures and time is not important in these models. Due to their time-inert property, we further group this type of models as “static model”. Actually most of these static models were proposed during the early phase of software reliability analysis (Lyu, 1996) and they failed to recognize the stochastic behaviors of software failure. Only a few of these models are still in practical use and they mainly serve as a tool for prior estimation of software failures (Lyu, 1996). In this thesis, only a few selected models from this group will be reviewed.

Bayesian models focus on the information about the software before software testing and combine with the collected data to make a more accurate estimation and prediction of the reliability.
If we define a process \( \{N(t), t \in T\} \) where \( N(t) \) denotes the total number of software failures encountered up to time \( t \) and \( T \) denotes the time interval, then we could see that the software failure process can be studied as a stochastic process. With plenty of theoretical support in stochastic process theory, many authors have proposed stochastic software reliability models. According to different processes that are modeled in their work, stochastic SRMs could further be divided into two sub-groups, the Markov group and the Non-Homogeneous Poison Process (NHPP) group. Since these models mainly focus on the software testing phase in which the reliability of software is continuously increasing as the testing continues, they are also called Software Reliability Growth Models.

A software reliability model belongs to the Markov group if its probabilistic assumptions of the failure counting process are essentially a Markov process. The main characteristic of a Markov model is that the software at a given time point is at a given state, which represents the remaining faults within the software system.

The non-homogeneous Poisson process models are the most widely used ones in both industry and research. Just as the name indicates, the software failure process is described using a non-homogeneous Poisson process.

We group our reviewed work according to their probabilistic assumptions. The classification is shown in the following structure.
Chapter II. Literature Review

Figure 2-1 Software Reliability Models Classification

Static Models

All software failures are associated with existing software faults. Software testing cannot eliminate all software faults and it is hard to directly deduce the total number of software faults from the software testing data.

A traditional statistical sampling technique called “capture-recapture” sampling was applied to predict the total number of software faults within a software system during the early 1970s and was systematically reported by Schick and Wolverton (1978). It is called the Fault-Seeding model. In this model, it is assumed that $M$ known numbers of faults, called “seeded” faults, are inserted into the software and are to be detected during software testing. Suppose that during testing, $k$ faults are detected and $m$ of them are recognized as seeded faults. Hence the number of initial faults that are detected is $k - m$. In this model, it is further assumed that the
Chapter II. Literature Review

Initial faults and the inserted “seeded” faults are equally likely to be detected. Based on such an assumption, we can estimate the total number of initial faults within the software system as

\[ N = \frac{M(k - m)}{m} \] (2.1)

The faults-seeding model is actually “borrowed” from zoology scientists who deal with animal population census. There are modifications of the basic fault-seeding model (Knight and Ammann, 1985) but the accuracy of these models is often doubtful (Lyu, 1996) because the assumption of “being equally likely to be detected” is too strong. The accuracy of faults-seeding models depend highly both on the seeded faults and the test case (Lyu, 1996) and such drawbacks have restrained the application of this model.

Bayesian Models

Bayesian models are the earliest software reliability models in the existing literature that systematically defined and derived the reliability of software systems. They are proposed mainly on the assumption that software failure data could be analyzed with the experiences gained from previous releases. One of the most well-known Bayesian model is the L-V model, which was proposed by Littlewood and Verrall (1973).

In the L-V model, time between failures, \( t_i, i = 1,2,3,\ldots, n \) is assumed to be independent and exponentially distributed:

\[ f(t_i \mid \lambda_i) = \lambda_i \exp\{-\lambda_i t_i\} \] (2.2)
where $\lambda_i$ is an unknown parameter and its prior distribution density function is to be obtained from the knowledge gained from previous releases of the software. Specifically, in L-V model, the authors assume that the parameter follows a Gamma distribution with parameters $\alpha$ and $\psi(i)$

$$f(\lambda | \alpha, \psi(i)) = \frac{[\psi(i)]^\alpha \lambda_i^{\alpha-1} \exp\{-\psi(i)\lambda_i\}}{\Gamma(\alpha)}$$  \hspace{1cm} (2.3)$$

In the above equation, $\alpha$ is the shape parameter and $\psi(i)$ is the scale parameter, which represents the number of detected faults. By constructing the likelihood function with the time series data, we could obtain the estimated values of $\alpha$ and $\psi(i)$. And the current software reliability can be expressed as

$$\hat{R}(t) = \left[ \frac{\hat{\psi}(i)}{t_i + \hat{\psi}(i)} \right]^\hat{\alpha}$$  \hspace{1cm} (2.4)$$

Due to the mathematical tractability, Bayesian models used to gain much popularity among researchers, especially in the early phase of software reliability research when more emphasis is on mathematical modeling. There are many modifications and extensions of the L-V model and there is also another famous branch of Bayesian models called Langberg and Singpurwalla model (Langberg and Singpurwalla, 1985). However, since this type of model usually requires much computational effort and it does not usually provide better results when
compared with other models, authors have been gradually moving their focus from this group of models to others.

**Markov Models**

If we regard the software failure process as a counting process and assume successive software failures are independent of each other, then the time to next failure given that a failure has happened only depends on the time at which the latest failure happened. In mathematical terms, the process satisfies the Markov property. The failure process can thus be studied under the Markovian framework and models under this frame are called Markov SRMs. These models are effective in describing the fault-removal processes.

Generally, in a Markov SRM, the following assumptions are made (Shanthikumar, 1981):

(i) The number of initial faults within the software is fixed but unknown

(ii) Once a software failure is encountered, the corresponding software fault is removed immediately with certainty.

(iii) Times between failures are independent, exponentially distributed quantities.

(iv) All remaining faults in the software contribute the some weight to the software failure intensity. The weight proportionality is a function of time.

If we assume that there are $N_0$ initial faults within the software, then according to Shanthikumar (1981), the failure intensity $\lambda(n,t)$ at time $t$ given that $n$ failures have happened and removed is:

$$
\lambda(n,t) = \phi(t)(N_0 - n), n = 1,2,3,...,N_0
$$

(2.5)
where \( \phi(t) \) is the proportionality function. The time between the \( i^{th} \) and \((i+1)^{th}\) failures is distributed with the density function:

\[
f(t_i \mid \lambda_i) = \lambda(i,t) \exp\{-\lambda(i,t)t_i\}
\]  
(2.6)

With sufficient historical data, we can estimate the parameters \( N_0, \phi \) using Maximum Likelihood Estimation (MLE) method with the likelihood function

\[
L(t_1, t_2, ..., t_n; N_0, \phi) = \phi^n \prod_{i=1}^{n} (N_0 - i + 1) \exp\left\{-\phi \sum_{i=1}^{n} (N_0 - i + 1)t_i\right\}
\]  
(2.7)

and then we are able to predict the expected time to next failure. Software reliability is derived as:

\[
\hat{R}(x \mid \hat{\lambda}) = \exp\left\{-\hat{\lambda}(i,t)x\right\} \quad \text{where } x \text{ is the elapsed time}
\]  
(2.8)

More specifically, if we assume the proportionality is a constant function, i.e, \( \phi \), and does not change with time, then we can replace \( \phi(t) \) with \( \phi \) and we get the famous Jelinski-Moranda model (Jelinski and Moranda, 1972):

\[
\lambda = \phi(N_0 - n), n = 1, 2, 3, ..., N_0
\]  
(2.9)
where $\phi$ can be comprehended as the average weight of failure intensity that each fault contributes. It can be clearly observed that equation (2.9) is a strictly decreasing function of $n$, which means the failure intensity is decreasing and the expected time to failure increases, thus software reliability is increasing. Such failure intensity function is called Decreasing Failure Intensity (DFI) function and more DFI examples are discussed by Xie (1990), such as the power-type function (Xie and Bergman, 1988), which is a direct generalization of the JM model. As an early group of SRMs, the Markov models had been successfully applied in some industrial cases (Jelinski and Moranda, 1972). However, this type of model suffers criticism due to its inaccuracy of parameter estimation. The estimation accuracy of parameters heavily depends on the historical data and sometimes estimation errors are unacceptable.

Non-Homogeneous Poisson Process Models

The software failure process can also be modeled as a non-homogeneous Poisson process (NHPP). NHPP was firstly widely used in modeling and analyzing hardware reliability and then it was adapted to model software reliability. It is now the most popular group of SRMs. There are many proposed NHPP models, which are based on different assumptions and focus on different aspects of software testing. In all, all NHPP models follows the following basic assumptions (Xie, 1991; Lyu, 1996; Pham, 1999):

(i) A software system is subject to failure at random caused by software faults.

(ii) No failures are experienced at the beginning of testing.

(iii) The probability that a failure will occur in a time interval $[t, t + \Delta t]$ is $\lambda \Delta t + O(\Delta t)$, where $\lambda$ the failure intensity, which may depend on $t$. 

18
Chapter II. Literature Review

(iv) The probability that more than one failure occur simultaneously is assumed to be zero.

NHPP models further assume that the number of software failures during a time interval $\Delta t_i = t_{i+1} - t_i, i = 0, 1, 2, ..., n$, follows the Poisson distribution:

$$\Pr \{ N_i = n_i \} = \frac{[m(t_{i+1}) - m(t_i)]^{n_i}}{n_i!} \exp \{ m(t_{i+1}) - m(t_i) \}, i = 1, 2, \ldots, n$$ (2.10)

where $m(t)$ is often called the mean value function (MVF) and represents the total number of expected software failures experienced up to time $t$. From equation (2.10) we can see that if a MVF is obtained, the corresponding NHPP model can also be determined. What’s more, MVF is easy to understand and it has a clear physical meaning $m(t) = E[N(t)]$. Thus, a MVF is usually used to describe an NHPP SRM. An MVF is defined as:

$$m(t) = \int_0^t \lambda(x) dx$$ (2.11)

where $\lambda(x)$ is the failure intensity at time $x$. We can predict the software reliability given that we have already known the MVF. The software reliability in the time interval $[t, t + x]$ can be calculated as:

$$R(x \mid t) = \exp \{ m(t) - m(t + x) \}$$ (2.12)
Generally speaking, most NHPP models have an underlying assumption that the failure intensity is proportional to the residual faults content in the software (Pham, 1999). If we denote $a(t)$ as the total number of fault in the software at time $t$ (including detected and undetected ones), $b(t)$ as the fault detection rate which reflects the efficiency of testing, the failure intensity function could be written as:

$$\frac{\partial m(t)}{\partial t} = \lambda(t) = b(t)[a(t) - m(t)]$$  \hspace{1cm} (2.13)

By combining equation (2.12) and (2.13), we can obtain a general form of MVF of NHPP models:

$$m(t) = e^{-B(t)} \int_0^t a(\tau)b(\tau)e^{B(\tau)} d\tau$$  \hspace{1cm} (2.14)

where $B(t) = \int_0^t b(\tau)d\tau$. The above general form of NHPP models is first introduced by Pham (1999). By adopting different forms of $a(t)$ and $b(t)$, we can get different NHPP models. These different NHPP models can be further classified according to the value of the limit of their MVF. If $\lim_{t \to \infty} m(t) = \infty$, the corresponding NHPP model could be classified as infinite failure model; otherwise it could be classified as finite failure model (Xie, 1991; Lyu, 1996). Since the limit of $m(t)$ stands for the expected total number of failures that the system would eventually encounter, this classification scheme is meaningful to software practitioners.
If specific forms of $a(t)$ and $b(t)$ are assigned, we get specific NHPP SRMs. If $a(t) \equiv a$ and $b(t) \equiv b$, the famous Geol-Okumoto model (Geol and Okumoto, 1979) is derived:

$$m(t) = a(1 - e^{-bt})$$

(2.15)

In this model, constant debugging rate and perfect debugging is assumed.

If $a(t)$ remains constant while $b(t) = \frac{b^2 t}{1 + bt}$, the delayed-S shaped model (Yamada and Osaki, 1984) is derived:

$$m(t) = a\left[1 - (1 + bt)e^{-bt}\right]$$

(2.16)

In this model, the scenario that failures detected in later phase of testing are harder to remove is modeled.

If $a(t)$ remains constant while $b(t) = \frac{b}{1 + \beta e^{-bt}}$, the inflexion-S shaped model (Ohba, 1984) is derived:

$$m(t) = \frac{a(1 - e^{-bt})}{1 + \beta e^{-bt}}$$

(2.17)

In this model, Ohba believes that some faults in a software system are mutually dependent and such dependence is not restricted to happen only at the beginning of testing. More
different forms of $a(t)$ and $b(t)$ is discussed and compared by Pham (2003). NHPP models are mathematically more tractable than other groups of SRM.

Unification of Markov Model and Non-homogeneous Poisson Process Model

Due to the stochastic nature of NHPP and Markov SRMs, it is possible to unify them under a larger framework. Such work is done by Gokhale et al (2004). A Non-Homogeneous Continuous Time Markov Chain (NHCTMC) is presented to model the failure process of software and most existing SRMs are unified under this framework. Related works can also be found (Gokhale et al, 2006; Gokhale and Lyu, 2005). Depending on the definition of states, models can be further grouped as pure death NHCTMC or pure birth NHCTMC. For example, the JM (Jelinsky and Moranda, 1972) model could be described as a pure death NHCTMC and most NHPP (Xie, 1991; Lyu, 1996; Pham, 2003) models could be deemed as a pure birth NHCTMC.

2.1.2 Software Reliability Model Extensions

Basic software reliability models are reviewed in the previous section and these models show how people systematically analyze software reliability problems from a quantitative point of view. However, most of these models were proposed decades ago and researchers in this field are shifting their focus from basic modelling to extending reliability models so that they can cope with more complex situations.

In many SRMs, it is assumed that software failures occur randomly and are removed with certainty when they occur. However, this is a simplification of the real world situation and often
software failures are hard to remove. In fact, the problem of imperfect debugging commonly exists in industrial practice and newly introduced faults also have apparent impact on the target software system (Fujiwara and Yamada, 2003).

Research efforts have been devoted to incorporate the imperfect debugging scenario into software reliability modelling (Pham et al, 1999; Lee et al, 2002; Cai et al, 2010; Chang and Liu, 2009). Basically, to incorporate imperfect debugging into software reliability models, it is commonly assumed by many authors that each software failure removal action has certain probability to introduce a new software fault.

Pham (1993) first introduced a systematic extension of NHPP SRMs to incorporate imperfect debugging. He assumed if detected faults are removed, then there is a possibility to introduce new faults with a constant rate $\beta$, which can be expressed as

$$\frac{\partial a(t)}{\partial t} = \beta \frac{\partial m(t)}{\partial t}$$

(2.18)

By extending the general form of NHPP models of equation (2.13), we get

$$m(t) = \frac{a}{1-\beta}(1-e^{-(1-\beta)bt})$$

(2.19)

By comparing equation (2.19) with the original general form of NHPP models of equation (2.14), we can see a clear impact of imperfect debugging. By following the work of Pham (1993), many researchers also focused on this topic (Bhaskar and Kumar, 2006; Gokhale
et al, 2006; Shyer, 2003; Tokuno and Yamada, 2003; Chatterjee et al, 2004; Xie and Yang, 2003; Chang and Liu, 2009) with fruitful results.

Other than taking the scenario of imperfect debugging into consideration, researchers are also working in other ways to help build more precise software reliability models. Software testing efforts is the most popular type. Software testing effort is first introduced to model the software development effort (Huang and Kuo, 2002; Huang et al, 2007). Most of them are parametric because they predict development effort using a formula of fixed form that is parameterized from historical data records. During the software testing phase, many testing efforts, such as the man power, the number of executed test cases, and the CPU time, are consumed. Typical testing effort functions are the exponential curves, Rayleigh curves, Weibull curves (Huang et al, 2007) and the logistic curves (Huang and Kuo, 2002). By selecting different testing effort functions that represent different testing strategies, we are able to construct more competent software reliability models that better models the failure process.

To incorporate testing effort into software reliability modeling, it is assumed that the software failure removal process follows NHPP and the mean number of faults detected in the time interval by the current testing-effort is proportional to the mean number of remaining faults in the system (Huang and Kuo, 2002; Huang and Lyu, 2005; Kapur et al, 2007; Huang et al, 2007). By denoting the testing effort rate function as \( w(t) \), the above assumptions can be expressed using the following differential equation:

\[
\frac{\partial m(t)}{\partial t} \times \frac{1}{w(t)} = r[a - m(t)]
\]

(2.20)
where \( r \) is a proportional factor. By solving the above equation, we obtain a new type of software reliability model that takes software testing efforts into consideration:

\[
m(t) = a\left[1 - \exp\left[rW(t) - rW(0)\right]\right]
\]

(2.21)

where \( W(t) = \int_{0}^{t} w(s)ds \) is the cumulative testing effort up to time \( t \).

By incorporating the testing effort into software reliability models, software practitioners can have more confidence in making decisions (Huang et al, 2007). For example, equation (2.21) can provide testers or developers with an estimate of the time needed to reach a given level of residual faults. It may also be used to determine the appropriate release time for the software to meet the expectations of customers.

### 2.2 Reviews on Software Availability Related Problems

Availability is an important measure of complex system performance. High availability is essential for safety critical systems, such as the control system in chemical plants, flight control system or systems in maritime industry. While in the last few decades, advanced technologies have been developed to enhance hardware reliability and availability (such as RAID, mirroring and redundant write cache, etc.) and there is plenty of literature in this field, little has been done in the aspect of software availability, which is also of importance to system availability.
As is covered in the previous section, the problem of software reliability originated from the internal software faults. The problem of software availability also has something to do with the internal software faults; however, it is also affected by many other factors. To determine whether a software system is available, people not only need to know the working state of the software system, but also how long it will take to fix a problem and how the software system reacts to external interference.

In this section, a systematic review on software availability related issues is presented. Being the main reason for software unavailability (Tokuno and Yamada, 2007), we first conduct a brief but thorough study on software maintenance first. Then we review the existing models for assessing software availability. In the end, the impact of software security issues on software availability is covered.

2.2.1 Software Maintenance and Software Availability

Software maintenance is the phase in the life cycle that includes all activity that occurs subsequently to the completion of the software development. Since software maintenance takes time and such time cannot be ignored when compared to software testing in the developing phase, there is a problem of software unavailability. As argued by many authors, software maintenance is the most important reason that makes software service inaccessible (Tokuno and Yamada, 2007). According to Lienzt and Swanson (1981), software maintenance focuses on either one or more of prevention, correction, perfection and adaptation. There are few works in the modeling or the analyses of software maintenance process since it is a multi-objective activity. Most of the
related work is extension of software engineering, which mainly deals with the methodologies in software maintenance. Topics in software maintenance are vast but we only focus on those that are related to this thesis. To be specific, we focus on the aspects of software maintenance that are related to software reliability and software availability. In other words, characteristics such as software maintenance methods and economic aspects (Tan and Mookerjee, 2005) will be reviewed in this section.

**Software Maintenance Methods**

People began to talk about how a system should be designed to improve maintainability as early as in the 1980s (McClure, 1981). More recently, research has emphasized improving the internal structure of code without altering its external behavior. The differences between methods of software maintenance basically derive from the maintenance modelling formalisms of software. “Maintenance modelling formalism” refers to the analysis of software maintenance from the view point of how the software was designed and constructed. To be specific, modeling formalisms, which refer to process, data and object-oriented modeling, are used to capture and represent the target system (Ahmed, 2006). Object-Oriented formalism, the 3rd Generation Language formalism and the UML formalism are the most popular formalisms.

Among the above mentioned modeling formalisms, Object-Oriented formalism is often advocated by computer science practitioners for their efficiency and ability of representativeness in both the designing and implementing phase. Darcy and Palmer (2006) argued that Object-Oriented formalism is also a better choice for software maintenance. Prior knowledge leads them to the assumption that projects using Object-Oriented formalism will lead to higher maintainability. However, Eierman and Dishaw (2007) prepared an experiment and argued that
Object-Oriented language and other 3rd generation language have their own advantages, respectively. Researchers also investigated the UML formalism (Arisholm et al. 2006, Lutters and Seaman, 2007) and advocate its ability in enhancing the maintainability of software documentation. Bandi et al. (2003) empirically explore the relationship between design and maintenance in object oriented design. Three complexity metrics—interaction level, interface size, and operation argument complexity—are proposed to quantify the impact of design on the future maintenance effort.

Other than discussing the modelling formalisms of software maintenance, practical maintenance methods are also discussed by many authors. Beyond software design, maintenance can be improved if certain maintenance practices are used (Tan and Mookerjee, 2005), e.g., change management procedures (Glass and Noiseux, 1981), predicting and prioritizing maintenance requests (Munson, 1981), guidelines for modifying and testing software (Yourdon, 1980), and implementation and integration of IE methodologies, CASE tools, and JAD techniques (Andrews and Leventhal, 1993). Kelly (2009) presented a detailed case study of the long-term evolution (maintenance) of three different industrial scientific software systems. The target software was released decades ago but it is still actively maintained and used.

Mathematical modelling is also adopted in dealing with software maintenance methods. Recently, Tsai and Chang (2003) provide a Markov model to find the optimal number of preventative maintenance activities so as to minimize the cost over an infinite horizon. Their model is based on a Markov decision process where the system takes on multiple states with each state corresponding to a different failure rate. Hewett and Kijsanayothin (2009) used a machine learning approach for predicting the software repair time in software maintenance. Different machine learning methods are applied and compared, and their case study showed good
prediction accuracy. Wang and Arisholm (2009) presented results from a quasi-experiment that investigates how the sequence in which maintenance tasks were performed affected the time required to perform them and the functional correctness of the changes made. Their conclusion is that the time spent on making the changes is not affected significantly by the task order of the maintenance tasks but the functional correctness is. Kung (2004) proposes and empirically validates a non stationary probability model for maintenance requests, where each type of maintenance request has a certain probability distribution that evolves over time. Bai et al (2008) studied the remaining faults of a software system and proposed the concept of Remaining Software Defect Estimation (RSDE) curve. Some theoretical and application issues of the RSDE curves are discussed.

An important aspect of managing maintenance is to predict and prioritize maintenance requests. Researchers have also put much effort on this topic. Burch and Kung (1997) empirically investigate the evolving behaviors of maintenance requests for a large software application. Four stages are identified; in each stage, there is a dominant type of request, for example, user support, repairing, and enhancement. Gefen and Schneberger (1996) provide a detailed case study of software modifications. They find the maintenance period is not one homogenous one but rather, consists of three distinct phases: stability, improvement, and expansion. It is also reported that the percentage of modifications that are a product of previous modifications can be as high as about 30 percent. Yu and Ramaswamy (2009) talked about the stability problem in software maintenance and software design and a model based on software version differences is presented to measure the evolutionary stability of software modules. Yu and Schach (2008) report the scenario of change propagation in software maintenance. Change propagation refers to the case that when changing one module in maintenance, maintainers
usually also need to change some other related modules in order to make the total system function correctly. An association-rule mining is conducted on three open source software systems and it shows that change propagation commonly exists in software maintenance. Hassan and Holt (2004) study propagation effects in maintenance to predict how implementing one batch of maintenance requests could lead to future requests. Cleland- Huang et al. (2003) propose a way to anticipate future requests so that these can be implemented accordingly. For a given set of change requests, ways of prioritizing these requests have also been proposed. Aversano and Tortorella (2004) study how business requirements affect system maintenance and propose attributes that can be used to assess the importance of change activities.

Software maintenance policies are also discussed under the framework of software maintenance methods. Feng et al (2006) conducted a study on two software maintenance policies: time-based maintenance and work-based maintenance and they provide insights into the management of software maintenance projects. In time-based maintenance, duration of each maintenance activity is fixed and the work that could be done in that duration is random while in work-based maintenance, the amount of work to be done in each maintenance activity is fixed and the time needed to complete the work is random. It is worth noticing that in their models (both the time-based and work-based), users request maintenance and the model decides when to conduct the maintenance. A Poisson process is used to model the arrival of requests so it is quite obvious that they are dealing with stable software and the maintenances should be regular adaptive or perfective. Sahin and Zahedi (1999, 2001) discussed warranty, maintenance and upgrade of software systems. In Sahin and Zahedi (1999), the authors develop a Markov decision process model for warranty, maintenance and upgrade of software systems, using the index of customer satisfaction as the state variable. Then they further present a framework to analyze
warranty, maintenance and upgrade decisions for software packages under different market conditions (Sahin and Zahedi 2001).

Cost of Software Maintenance

The fundamental reason why software maintenance attracts so much attention is that software maintenance is costly (Kan, 2003). Basically speaking, software maintenance is driven not only by the request of software users but also constrained by the software maintenance budget (Feng et al, 2006).

The cost of software maintenance is associated with the amount of work required, the cost for setting up maintenance and the cost incurred when software system is unable to perform the designed job (Feng et al, 2006). Some researchers treat the problem of the cost of software maintenance from the view point of economy (Banker and Slaughter 1997) but more just focus on the cost itself. Many empirical models for estimating software cost have been proposed (Banker and Kemerer, 1989, Banker, Davis, and Slaughter 1998) to represent the cost of software maintenance.

Some authors also try to estimate software maintenance cost by estimating the effort consumed in software maintenance. Ahn et al. (2003) propose a software maintenance project effort estimation model (SMPEEM) that uses function points to calculate the effort of planned maintenance activities. Fioravanti and Nesi (2001) propose a general model for adaptive maintenance effort prediction for object oriented systems. More detailed models of maintenance effort estimation have separated the fixed effort from the variable effort needed for software maintenance (Sneed, 2004).
Chapter II. Literature Review

Although there are many different approaches for estimating software maintenance cost, an interesting issue in dealing with software maintenance cost is that authors are eager to find whether there is an optimal batch size for software maintenance (Banker and Kemerer, 1989). There are many case studies to determine the optimal software maintenance batch size (Banker and Kemerer, 1998, Feng et al, 2006), however, these work are based on case-to-case scenarios and no author has ever proposed any systematic approach for deciding the optimal batch size for any general cases.

Chan et al (1996) summarize different kinds of software maintenance cases and offer several suggestions for conducting software maintenance:

1. use configuration management and change request procedures,
2. emphasize standards in maintenance as in development,
3. create a budget for maintenance,
4. align maintenance rewards to organizational performance, and so on.

2.2.2 Software Availability Assessment

The concept of software reliability may be abstract for ordinary people, but most people have experienced problems such as computer systems being unavailable to provide service. In other words, software availability directly affects how customers perceive a software product. While the problem of software reliability has been widely studied by many authors during the past few decades, the problem of software availability has received little attention. Xie (1991) concludes two reasons why there are few software availability applications: 1) software availability is harder do model because it is harder to trace every up and down state of a software
system explicitly and 2) reliability is of concern to more professionals while availability is of concern to more software users. However, with the development of computer hardware industry, the focus of software industry is changing from computationally effectiveness to human friendliness (e.g., changing from command line console to graphical user interface) and more authors are arguing that the quality of a software system should be perceived from the view point of software users (Takuno and Yamada, 2007). Although still limited, the problem of software availability is concerned by more and more software practitioners.

The first work about software availability was reported in the 1970s. Trivedi and Shooman (1975) first analyzed the problem of software availability using a Markov chain approach. According to Trivedi and Shooman (1975), the states of a software system can be divided into up and down state, depending on whether the software is functioning or not. The target software system is assumed to be in an up state in the very beginning. When a software failure occurs, the system is shut down and enters the down state. The fault which has caused this failure is then detected and removed before the system begins to function again. Both the functioning times and repair times are assumed to be random quantities and the process is modeled by a Markov process.

Denote $\lambda(t)$ as the transition rate from up to down state and $\mu(t)$ as the transition rate from down to up state. $p_i(t), i = n, n-1,...$ represents the probability that the software is in the $i$th functioning state at time $t$. Similarly, $q_i(t), i = m, m-1,...$ represents the probability that the software is in the $i$th down state at time $t$. The state probabilities can be obtained by solving the following differential equations:
\begin{align}
\frac{\dot{p}_n(t)}{\partial t} &= -\lambda(t) p_n(t) \\
\frac{\dot{p}_{n-i}(t)}{\partial t} &= -\lambda(t) p_{n-i}(t) + \mu(t) q_{m-i+1}(t) \quad i = 0,1,2,\ldots \\
\frac{\dot{q}_{m-i}(t)}{\partial t} &= -\mu(t) q_{m-i}(t) + \lambda(t) p_{n-i}(t)
\end{align}

Together with the initial conditions:

\begin{equation}
p_n(0) = 1, p_k(0) = 0, \text{ for } k \neq n \tag{2.23}
\end{equation}

The solution to the above equations system is given by

\begin{equation}
p_{n-i}(t) = \left(\frac{\lambda \mu}{\mu - \lambda}\right)^i e^{-\lambda t} \sum_{j=0}^{i-j} t^{i-j} \frac{(-1)^{i-j+1} c_{ij} e^{-(\mu-\lambda)t} + (-1)^j d_{ij}}{\mu - \lambda(i-j)!} \tag{2.24}
\end{equation}

where

\begin{equation}
c_{ij} = \begin{cases} 
0, & \text{for } j = 0 \\
1, & \text{for } j = 1 \\
\binom{i+j-1}{j-1}, & \text{otherwise}
\end{cases}
\end{equation}

and
Trivedi and Shooman (1975) treat software availability as the probability that the software is operating at time $t$. This probability can be calculated as the summation of all the state probability that the software is in the up state:

$$A(t) = \sum_{k=0}^{\infty} p_{n-k}(t) \quad (2.25)$$

The model discussed above is easy to understand, however, it requires lots of computational effort to solve. Kim et al (1982, 1984) further extended Trivedi and Shooman’s model by incorporating imperfect debugging issues. Takuno and Yamada (2003) also extended this model by taking software restorations into consideration.

While some authors treat software availability as certain type of probability, some authors (Gokhale et al, 2004; Zhang and Pham, 2002) proposed another method for estimating software availability. In their work, software availability is regarded as the percentage of total scheduled service time when systems are operational and ready to provide service. By denoting the mean time to failure as MTTF and mean time to repair as MTTR, the software availability can be simply derived as

$$A(t) = \frac{MTTF}{MTTF + MTTR} \quad (2.26)$$
In software quality analysis, the software failure process can be uniquely modeled by MTTF, which was covered in the previous section. However, there is little work on how to estimate the MTTR of the software maintenance process and Gokhale et al (2004) just assume MTTR is a known constant, which cannot be validated from our previous review on software maintenance. More research is needed in the modelling of MTTR in this type of model.

The difference between equation (2.25) and (2.26) derives from the difference in the definition of software availability. In the next chapter we will prove that the two definitions are actually equivalent.

### 2.2.3 Malicious Software and Its Impact on Software Quality

The development and wide spread of computer network (e.g., internet, Local Area Network, Wide Area Network, etc) has made our life much easier by bringing us more convenience in communication and work processing, however, it also brings in more threats for computer software systems than ever before – that is, the threat of malicious software.

Malicious software, which is more often called malware for short, is software designed to infiltrate or damage a computer system without the owner's informed consent (e.g., viruses, backdoors, spyware, Trojans, and worms) (Ye et al, 2010). According to Giffin (2010), it is estimated that about 15% of internet-connected computers are infected by certain types of malicious software, and this figure is just a rough estimation and the true rate may be as high as 25%. The threat of malware is more severe than ever before during this network age is because that malware not only hampers normal functioning of the target victim, but also keeps trying to re-produce itself. Giffin (2010) also report that infected systems flood the Internet with unwanted
traffic that includes email spam, denial-of-service (DoS) attacks, illegal content, and continued malware propagation to new victims. Telang and Wattal (2008) further report the fact that software vulnerability have great impact on stock price, which is an essential concern of capital holders.

As of its malicious nature, malware can greatly degrade the performance of infected systems and a lot of problems arise (Ye et al, 2010), such as degradation of software reliability and software availability, as well as higher software maintenance cost (Kondakci, 2009).

The first step that people deal with malware is to identify the malware. In the early days of anti-virus software industry, anti-virus software vendors used a special technique called “signature-based” method to identify and eliminate existing viruses. A signature is a short string of bytes which is unique for each known malware so that future examples of it can be correctly classified with a small error rate. This signature-based method used to be effective, however, with the increase of types of malware and rapid evolution and mutation of malware (Gutpa et al., 2009, Feng and Gutpa, 2009), this classic signature-based method always fails to detect variants of known malware or previously unknown malware, because the malware writers always adopt techniques like obfuscation to bypass these signatures (Sung et al. 2004).

In order to overcome the disadvantages of the widely-used signature-based malware detection method, data mining and machine learning approaches are proposed for malware detection (Kolter and Maloof 2004; Schultz et al. 2001; Sung et al. 2004; Wang et al. 2003; Christodorescu et al. 2007; Ye et al. 2007). Neural Networks as well as immune system are used by IBM for computer virus recognition (Tesauro et al. 1996). Naive Bayes, Support Vector Machine (SVM) and Decision Tree classifiers are used to detect new malicious executables based on small data collection in the previous studies (Kolter and Maloof 2004; Schultz et al. 2001; Sung et al. 2004; Wang et al. 2003; Christodorescu et al. 2007; Ye et al. 2007).
2001; Wang et al. 2003). Associative classification (Liu et al. 1998), as a new classification approach integrating association rule mining and classification, becomes one of the significant tools for knowledge discovery and data mining (Thabtah 2007). More recently, Ye et al (2010a, 2010b) proposed a new approach for determine malicious software from a large scale gray list.

The defense against malware largely depends on malware defense software. Rather than talk about techniques in defending malware, recent research focuses on the strategy of deploying malware defense software systems. Surisetty and Kumar (2010) provide a detailed report of the testing results of McAfee Security Center software and Larsen (2009) further presents an empirical investigation of factors affecting small- and medium-sized business (SMB) executives' decision to adopt anti-malware software for their organizations. A research model was developed by adopting and expanding the protection motivation theory from health psychology, which has successfully been used to investigate the effect of threat and coping appraisal on protective actions. This study demonstrates that threat and coping appraisal successfully predict SMB executives' anti-malware software adoption intention, leading to SMB adoption. In addition, considerable variance in adoption intention and actual SMB adoption is addressed by social influence from key stakeholders and situation-specific variables, such as IT budget and vendor support. Vendor support was a key facilitator of the anti-malware adoption for information system experts and IT intensive industry groups, while IT budget was for non-information system expert and non-IT intensive industry groups.

An important characteristic of malware is that it tries to reproduce itself whenever and wherever possible. This epidemic behavior also attracts many authors to investigate the propagation of malware. Kondakci (2008) presents a discrete-time Markov chain to model the virus transmission between different nodes among the network and provides a numerical analysis
which is based on a malware epidemic model of email spread. Lelarge (2010) model and quantify the impact of malware externalities on the investment in security features in a network. They study a network of interconnected agents, which are subject to epidemic risks such as those caused by propagating viruses and worms. Each agent can decide whether or not to invest some amount to self-protect and deploy security solutions which decreases the probability of contagion. Wang et al. (2009) introduce the concept of time delay to represent the time take to re-assembly the system and deploy anti-virus software. Based on the concept of time delay, they propose a differential equation model for the propagation of software virus. Song and Jiang (2010) take the nodes with different anti-attack abilities in scale-free networks into consideration and investigate the probabilistic behaviors of malware propagation in scale-free complex networks.

The focus of existing literature on malicious software is more on its technical aspect and few efforts are found in dealing with the quality problems of malicious software-infected software systems, which spurs us to further investigate this topic in this thesis.
Chapter 3  Software Availability Modelling and Application Extensions

In this chapter, a unified framework for modelling software availability is proposed. This approach takes the stochastic behaviors of both software failure and maintenance process into consideration and yields satisfactory results. It is also shown in this chapter how the proposed model can be applied to industrial practice.

3.1 Introduction

Availability is an important measure of system performance. High availability is essential for safety critical systems, such as the control system in chemical plants, flight control system or systems in maritime industry. While in the last few decades, advanced technologies have been developed to enhance hardware reliability and availability (such as RAID, mirroring and redundant write cache, etc.) and there is plenty of literature in this field (Kozlov et al, 2008), little has been done in the aspect of software availability, which is also of great importance to system availability (Kan, 2003). Most of the research efforts of many software practitioners have been focused on software reliability. Many influential results (Xie, 1991; Pham, 2003; Lyu, 1996) have been reported. However, reliability is a rather conceptual item for software customers and the main and direct concern of customers is that software service should be available whenever it is needed (Tokuno and Yamada, 1999; Tokuno and Yamada, 2007). On the other hand, software availability also plays an important role in maintaining a high quality software system (Sha, 2001).
and serves as a quality indicator that is vital for decision making on many occasions. The importance of system availability, which consists of both hardware and software availability, is quite apparent and should be emphasized (Kan, 2003).

In many software reliability models, the debugging and repairing activities are assumed to be performed once a software failure is detected and it takes no time to complete the tasks. This assumption is called “instant debugging” and is used widely in the literature of software reliability research (Pham, 2003). However, in real industrial cases, it takes engineers much time to locate and fix bugs that cause software failures and the time for such work varies but cannot be ignored. When software systems are down for debug/repair, the services provided by software systems are no longer reachable and we regard it as ‘unavailable’.

Some authors consider software availability as the probability of software being in a working state at a given time in a given environment (Trevidi and Shooman, 1975) while others define software availability as the percentage of total scheduled service time when systems are operational and ready to provide service (Gokhale et al, 2004). Although the two definitions differ in forms, we show that the two actually can be transformed to each other in given conditions.

**Proposition:**

The probability definition of software availability and the proportion definition of software definition can be transformed to each other in given conditions.

**Proof Begins:**

41
Suppose a software system is to work for a period of $T$ time in a certain environment. At any time $t$ during the period $T$, the probability that the software system is working can be described by $a(t)$. The percentage of total scheduled service time when the system is operational is described by $A(T)$. $a(t)$ can be perceived as the availability definition by Trivedi and Shooman (1975) while $A(T)$ can be regarded as the availability definition of Gokhale et al. (2004). The following equation exists and the proposition is proved.

\[
E[A(T)] = \frac{\text{Expected total service time}}{T} = \frac{\int_0^T a(t)dt}{T}
\]

Proof Ends.

The remainder of this chapter is organized as follows: Section 3.2 presents the origination of the problem of software availability and a further investigation into software maintenance activities is conducted. Then the basic assumption together with a stochastic model which describes software operation and repair activities in an integrated manner is then proposed. An application extension which deals with optimal software maintenance policy under software availability constraints is then provided in Section 3.3. Section 3.4 summarizes this chapter with some discussions of the results.
3.2 Software Maintenance Activities and Software Availability

As discussed in the previous section, software availability is more a user-perceived term rather than a developer-perceived term and software availability is of concern after the software system is deployed. The problem of software availability derives from software repair activities. And in most cases, repair work is conducted in the form of software maintenance. In fact software maintenance includes many different types of work such as source code revision, document update and so on. It is hard to unify different forms of software maintenance work and model them in a universal representation. However, they all consume time and efforts. In this section, software maintenance activities are regarded as the main cause of software unavailability. They are investigated and decomposed into individual maintenance actions. Based on such decomposition, we propose an effort-based model to present the behavior of software maintenance activities and then the software operation is considered and finally a unified software availability model is obtained. Numerical examples are provided in the purpose of validating the effectiveness of our proposed models in quantitatively estimating software availability.

3.2.1 Decomposition of Software Maintenance Activities

People begin to pay attention to the problem of software reliability when the coding of a software project is done and the project shifts to testing phase. The main task in the software testing phase consists of two parts: testing and debugging. While the aim of software testing is to find as many software faults as possible, the aim of debugging is to fix the problems that are
encountered in testing. Most of the research on software reliability has focused on this phase and shares one common assumption when proposing models: fault-removal activities are instant and transient (Pham, 2003; Farr, 1996). Basically, in many real industrial cases, the tasks of testing and debugging are done by different teams of personnel during the software testing phase (Kaner et al, 1999). Testing and debugging are done on different but identical systems. Detected faults in testing are reported to the debugging personnel, who analyze and fix them, while testing keeps going without delay (Gokhale et al, 2006). It is reasonable to assume that the two activities of testing and debugging are parallel. There are no problems of availability during the testing phase since detected faults do not hamper software operation. It is also from this point of view that the instant-debugging assumption in most SRMs could be validated. Such phenomenon could be illustrated in the following figure.

![Testing and Debugging during Software Testing Phase](image)

*Figure 3-1 Testing and debugging during software testing phase*

After sufficient testing, software can be released and it shifts to the operation phase in the life cycle. It is nearly impossible to remove all the faults in the software in the testing phase, so software failures still occur in the operation phase. When software is released, testing activities
which are performed by testing personnel changes to normal operations that are performed by customers and fixing activities are carried out in the form of maintenance (Feng et al., 2006). However, software maintenance is conceptually different from software debugging in the testing phase for that the correction work need to be done exactly on the system where the problems are found but not on an identical backup system which is used in testing.

When a software system is deployed, customers are always expecting that the system is working under full capability all the time. When a software failure occurs, certain parts of or even the whole system would not work correctly. For the sake of safety and cost, customers would not allow encountered software failures being left unattended for long so they require maintenance. On the other hand, software maintenance could not be performed while the software is still in operation since software faults within the system need to be traced and the corresponding error in the source code need to be mapped and fixed (Hanebutte and Oman, 2005). We assume that the software system could not be online until maintenance is done. So it is no longer reasonable to assume that software operation and fixing are parallel and they should take place in alternative order. Examples can be found in manufacturing execution systems, real time control systems in chemical industry and so on. A demonstrative scenario can be decomposed in the following figure. In Figure 3.2, the two horizontal lines represent software operation and maintenance, respectively. Please be noted that this figure is just for illustration and each “maintenance line” does not necessarily mean only one maintenance activity is performed.
As can also be clearly seen from the above figure, there are apparent availability “gap” between software operations. These availability “gaps” are direct consequences of software maintenance activities. If proper estimation of each individual maintenance action can be obtained, the availability of the software system can be derived at ease.

3.2.2 Software Maintenance Efforts and Software Maintenance Modeling

Maintenance of software is one of the least-structured problems in information systems (Bhatt et al, 2006) and maintenance developers usually spend a large portion of time in understanding the target software system (Swanson, 1976). Based on the objective, software maintenance can be classified into three main categories: utilizing software support, source code change and documentation update (Chapin et al, 2001). Since source code change is the most common type of software maintenance and requires lots of work, in this section and without loss of generality, only the code change is considered when we model the software maintenance
process. However, due to the dynamic characteristics of modifying source code, it is hard to analyze the maintenance activities from traditional perspectives (Yu and Ramaswamy, 2009; Yu and Schach, 2008).

Huang and Kuo (2002) proposed the concept of testing efforts when trying to unify a number of different software reliability models and the documented results (Huang and Kuo, 2002; Huang and Lyu, 2005; Huang et al, 2007) showed that different modeling of the software failure process can be unified under the framework of testing efforts. Since the software failure process and software maintenance process are both share stochastic, as a result, we are motivated to study software maintenance activities from the maintenance efforts point of view. We regard maintenance effort as the driving power of maintenance activities and behaviors of individual maintenance participants can be described using the contribution of maintenance effort.

Maintenance process and effort modelling

Software maintenance effort usually refers to the effort in correcting software failures and can be comprehended as the cost of manpower, workload, money, etc. The maintenance of software systems involves many participants, who contribute personal effort in different ways. It is hard to monitor the exact contribution of an individual, and we assume that the total maintenance effort that each individual can provides in a unit time is the same. The total amount of maintenance efforts depends on the total number of active participants. In order to facilitate modelling the maintenance process, several assumptions are made.
Assumptions:

1) The expected amount of maintenance effort that individuals contribute is the same.
2) The amount of active maintenance participants depends on the state of maintenance.
3) Software failures are removed solely due to maintenance effort.
4) Maintenance rate is proportional to the maintenance effort consumption rate.
5) Maintenance rate is proportional to the number of remaining software faults.

Denoted as \( w(t) \), the maintenance effort rate at time \( t \) could be comprehended as proportional to the total number of active maintainers at time \( t \) and can be written as

\[
w(t) = \theta \cdot h(t)
\]  

(3.1)

where \( \theta \) is a constant coefficient and represents the maintenance effort contribution factor and \( h(t) \) is the total number of active maintenance participants at time \( t \).

A software maintenance process can be uniquely determined and represented by its mean value function. We denote the expected number of maintained software faults at time \( t \) as \( m(t) \), and this mean value function satisfies the following differential equation:

\[
\frac{dm(t)}{dt} = w(t)[\alpha - m(t)]
\]  

(3.2)

where \( \alpha \) represents the total number of software faults that can be eventually maintained.

Solving the above differential equation, we obtain
Chapter III. Software Availability Modelling and Application Extensions

\begin{equation}
\begin{split}
m(t) = \alpha [1 - \exp(-W(t))] \\
\end{split}
\end{equation}

where

\begin{equation}
W(t) = \int_0^t w(\tau) d\tau = \theta \int_0^t h(\tau) d\tau
\end{equation}

Judging from equations (3.3) and (3.4), we can see that the maintenance process mainly depends on the number of active maintenance participants. However, it is hard to monitor the exact number of participants and a close form description of \( h(t) \) is undesirable in most cases and needs to be estimated.

Maximum likelihood estimation is commonly used in such estimating processes. Our proposed model can be classified as a non-homogeneous Poisson process, and the probability that a maintenance event occurs during the time interval \([t, t + \Delta t]\) is described as:

\begin{equation}
p(\Delta t \mid t) = \exp[m(t + \Delta t) - m(t)]
\end{equation}

The likelihood function can be constructed as below:
\[
\ln L = \ln \left\{ \prod_{i=1}^{k} \frac{[m(t_j) - m(t_{i-1})]^n \exp (m(t_j) - m(t_{i-1}))}{n_i!} \right\} \\
= \sum_{i=1}^{k} n_i \ln \alpha - \alpha (1 - \exp \left[ -\theta \int_{0}^{t_i} h(\tau) d\tau \right] - \sum_{i=1}^{k} \ln (n_i)! + \\
\sum_{i=1}^{k} n_i \ln \left\{ \exp \left[ -\theta \int_{0}^{t_i} h(\tau) d\tau \right] - \exp \left[ -\theta \int_{0}^{t_i} h(\tau) d\tau \right] \right\}
\]

For the estimation of each unknown parameter, we can take the partial derivatives of equation (3.6) with respect to these parameters and assigning the derivatives to zero. For example, if \( h(t) \) is linear with parameter \( a \) and \( b \), then we can get a closed form of the mean value function of the maintenance process by solving the equations below:

\[
\frac{\partial \ln L}{\partial \alpha} = \frac{\partial \ln L}{\partial \theta} = \frac{\partial \ln L}{\partial a} = \frac{\partial \ln L}{\partial b} = 0
\]

Analytical solutions for equations like the above are hard to obtain and such problems are usually solved by numerical methods. Another problem associated with the estimation process is that \( h(t) \) needs to be carefully selected in order to represent the actual pattern. In the next section, a numerical example is shown to validate our proposed model’s ability to effectively model the maintenance process.
3.2.3 Software Maintenance Modeling—A Numerical Example

In the previous section, a software maintenance process model was presented together with methods for estimating parameters of the proposed model. The more the historical data is provided, the more confident we are about the estimated parametric. However, software maintenance data are usually regarded as more confidential to many companies than testing data and are not open to public. This is also one of the major reasons why research on software maintenance is behind that on software testing (Yu et al., 2006). However, with the development of open source software, which provides open and detailed research data for academic researchers, the situation is changing.

Different from commercial software systems, open source software is usually developed outside companies – mostly by volunteers – and the development method is quite different from the standard methods applied in commercial software development (Ko et al., 2006; Gyimothy et al., 2005). Also unlike commercial closed source software systems, open source software projects commonly do not have a specialized maintenance team. Although differently organized, the maintenance processes of open source software and closed source software are similar. Maintenance requests are first submitted and recorded. Then the maintenance tasks are assigned, and finally when tasks are finished, and the finishing time is recorded, the task is closed. The main difference between these two is that there is usually a tight timetable and a plan for closed source software projects whereas the maintenance progress of open source software projects is largely due to the contribution of individuals from the developer community.

Another important issue in open source software maintenance is the rapid release of new versions. The huge number of different versions of some open source software projects makes
them hard to maintain them all. Since the newer versions usually include fixes of bugs and enhancements of previous versions and since they are free, users often switch to the newer version and less attention is paid to the older versions. The maintenance speed of older versions of open source software can be apparently slowed down whereas the maintenance process can be very active for the newly released versions due to the shift of interests of users and developers. This phenomenon has been reported by Yu (2006).

The most common scenario of open source software maintenance is described next. When a version of open source software is released, many interested users rush to experience and maintain the new product. Software failures are triggered and reported frequently, and then they are maintained. As more and more participants are involved, the maintenance process speeds up gradually. However, when a newer version is released, most of the active participants will move to the newer version and the maintenance of the older version is greatly slowed down. In order to model the peak phenomenon and sharp slope in the active software maintenance personnel changes, we use a Rayleigh function to model the expected number of active maintenance participants:

\[
h(t) = \frac{t}{\sigma^2} e^{-\frac{t^2}{2\sigma^2}}
\]

(3.7)

where \(\sigma\) is a Rayleigh parameters that need to be estimated.

Incorporating equation (3.7) into (3.3), we get
\[ m(t) = \alpha \{1 - \exp\left[-\theta \left(1 - \exp\left(-\frac{t^2}{2\sigma^2}\right)\right)\right]\} \]  

(3.8)

In order to see the ability of the proposed model of software maintenance effort and software maintenance process, parameters of equation (3.8) need to be estimated using field data to check its validity. In this thesis, we choose to use the maintenance data generated by Apache web server. The reason why it is chosen here is because it is, according to the Netcraft survey (2009), the most widely used web server with 100 million users and it has an open organization where a large number of developers have the rights to update and change files. Apache 2.0.35 was first available to the public on 2002/04/06. It is the first release of Apache’s major version 2.0. We select this release as our example. The retrieved maintenance data are presented in Table 3.1.
In this section, we use both maximum likelihood estimation (MLE) and least square estimation (LSE) to estimate the fitting performance. The results of estimation are presented in Table 3.2.

Table 3-2 Summary on Model Estimates and Goodness-of-fit for Apache Release 2.0.35

<table>
<thead>
<tr>
<th>Estimation methods</th>
<th>Estimate values</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLE</td>
<td>( \hat{\alpha} = 70.68 )</td>
<td>46.06</td>
</tr>
<tr>
<td></td>
<td>( \hat{\theta} = 4.66 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \hat{\sigma} = 9.30 )</td>
<td></td>
</tr>
<tr>
<td>LSE</td>
<td>( \hat{\alpha} = 66.83 )</td>
<td>30.76</td>
</tr>
<tr>
<td></td>
<td>( \hat{\theta} = 3.30 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \hat{\sigma} = 3.58 )</td>
<td></td>
</tr>
</tbody>
</table>
The results include the estimated parameters and the overall goodness-of-fit which is measured by Mean Square Error (MSE). It is shown that LSE yields a lower MSE than that of MLE. Table 3.3 provides the detailed fitting results. Table 3.3 includes actual number of corrected faults and the fitted values by MLE and LSE.

A graphic illustration of the fitted values is given in Figure 3.3. It is observed that LSE curve is very close to the actual curve in the intervals [0, 4] and [13, 27] while it is far from the actual curve in the interval [5, 10] and all the rest of time. On the other hand, MLE curve generally represents the pattern of the actual curve and its fit keeps improving with time and it is this set of estimation results that are used for further analysis.
### Table 3-3 Field Data with Fitted Data for Apache 2.0.35

<table>
<thead>
<tr>
<th>Weeks</th>
<th>Actual Corrected Number</th>
<th>ML Estimates</th>
<th>LS Estimates</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
<td>1.87</td>
<td>7.95</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>7.13</td>
<td>25.41</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>14.86</td>
<td>41.75</td>
</tr>
<tr>
<td>3</td>
<td>42</td>
<td>23.83</td>
<td>52.45</td>
</tr>
<tr>
<td>4</td>
<td>47</td>
<td>32.90</td>
<td>58.32</td>
</tr>
<tr>
<td>5</td>
<td>49</td>
<td>41.20</td>
<td>61.32</td>
</tr>
<tr>
<td>6</td>
<td>52</td>
<td>48.26</td>
<td>62.84</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>53.92</td>
<td>63.61</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>58.28</td>
<td>64.01</td>
</tr>
<tr>
<td>9</td>
<td>56</td>
<td>61.52</td>
<td>64.21</td>
</tr>
<tr>
<td>10</td>
<td>60</td>
<td>63.88</td>
<td>64.30</td>
</tr>
<tr>
<td>11</td>
<td>62</td>
<td>65.58</td>
<td>64.34</td>
</tr>
<tr>
<td>12</td>
<td>63</td>
<td>66.80</td>
<td>64.36</td>
</tr>
<tr>
<td>13</td>
<td>63</td>
<td>67.67</td>
<td>64.37</td>
</tr>
<tr>
<td>14</td>
<td>63</td>
<td>68.29</td>
<td>64.37</td>
</tr>
<tr>
<td>15</td>
<td>63</td>
<td>68.74</td>
<td>64.37</td>
</tr>
<tr>
<td>16</td>
<td>63</td>
<td>69.06</td>
<td>64.37</td>
</tr>
<tr>
<td>17</td>
<td>63</td>
<td>69.30</td>
<td>64.37</td>
</tr>
<tr>
<td>18</td>
<td>63</td>
<td>69.48</td>
<td>64.37</td>
</tr>
<tr>
<td>19</td>
<td>64</td>
<td>69.61</td>
<td>64.37</td>
</tr>
<tr>
<td>20</td>
<td>64</td>
<td>69.71</td>
<td>64.37</td>
</tr>
<tr>
<td>21</td>
<td>65</td>
<td>69.79</td>
<td>64.37</td>
</tr>
<tr>
<td>22</td>
<td>65</td>
<td>69.84</td>
<td>64.37</td>
</tr>
<tr>
<td>23</td>
<td>65</td>
<td>69.88</td>
<td>64.37</td>
</tr>
<tr>
<td>24</td>
<td>66</td>
<td>69.92</td>
<td>64.37</td>
</tr>
<tr>
<td>25</td>
<td>66</td>
<td>69.94</td>
<td>64.37</td>
</tr>
<tr>
<td>26</td>
<td>66</td>
<td>69.96</td>
<td>64.37</td>
</tr>
<tr>
<td>27</td>
<td>68</td>
<td>69.97</td>
<td>64.37</td>
</tr>
<tr>
<td>28</td>
<td>68</td>
<td>69.98</td>
<td>64.37</td>
</tr>
<tr>
<td>29</td>
<td>68</td>
<td>69.99</td>
<td>64.37</td>
</tr>
<tr>
<td>30</td>
<td>68</td>
<td>69.99</td>
<td>64.37</td>
</tr>
<tr>
<td>31</td>
<td>69</td>
<td>70.00</td>
<td>64.37</td>
</tr>
<tr>
<td>32</td>
<td>70</td>
<td>70.00</td>
<td>64.37</td>
</tr>
</tbody>
</table>
3.2.4 Validation of Software Maintenance Modelling

More information can also be derived from the estimation. As discussed in the above context, the maintenance pattern of open source software changes when a newer version is released. To be more specific, the maintenance efforts shift to the newer version when a newer version is released, and the maintenance process of the current version slows down. This phenomenon can be mathematically explained as the change point of maintenance rate. In other words, we can use our proposed model to predict the release time of the newer software version.
by taking the derivative of the maintenance rate function and make the derivative equal to zero. If the release time of the newer software version can be correctly predicted, then the feasibility of our proposed model in modeling software maintenance can also be demonstrated:

\[ O(t) = \frac{dw(t)}{dt} = \theta \frac{dh(t)}{dt} = 0 \]  

(3.9)

By substituting the estimated parameters in Table 3.2 into equation (3.9) and solving it, we obtain the change point for ML estimates as \( t = 4.2 \) weeks. According to actual data, the next release (Apache 2.0.36) was exactly one month after Apache 2.0.35. The prediction well complies with the actual data and the ability of our proposed model in successfully modelling software maintenance efforts and software maintenance process is also validated.

3.2.5 Software Availability Modeling Considering Software Maintenance

In previous sections, we investigated the problem of software maintenance activities and proposed a model for software maintenance effort and process. If we look back at Figure 3.1 and Figure 3.2, clear evidence could be found that the problem of software availability arises because of software maintenance. As is clearly shown in Fig.3.2, the software system is online for a certain proportion of scheduled service time, due to the down time caused by fixing the failures. We call two successive up state and down state as a service cycle. We define \( U(t) \) as the accumulated time spent in operation and \( D(t) \) as the accumulated time spent in repair in a given service period \( t \). Without loss of generality, we calculate software availability as the proportion
of time when software system is in up status and available for service. Define $A$ as software availability and $\overline{A}$ as software unavailability, then software availability and unavailability could be calculated with ease:

$$A = \frac{U(t)}{U(t) + D(t)}$$

$$\overline{A} = \frac{D(t)}{D(t) + U(t)}$$

where $U(t)$ is the arithmetic sum of subsequent “time to failure” and $D(t)$ is the arithmetic sum of subsequent “time to repair”.

To keep software systems functioning at their maximum capability while assuring acceptable level of software availability is the main objective. However, since the time to next failure and the time to repair of the next failure is random, in most cases software practitioners could not obtain figures of software availability with acceptable confidence. Researchers on most SRMs (Xie, 1991; Lyu, 1996; Pham, 2003) analyze the software failure process and model the time to failure, which enables us to predict the expected time at which the next failure would happen. Most of these models are either based on Markov chains or non-homogeneous Poisson process and Gokhale et al (2004) unified most of these proposed models which describe the software failure process as Non-Homogeneous Continuous Time Markov Chain (NHCTMC). Basically, the stochastic failure process which is described by NHCTMC, denoted by $\{X(t)\}$, counts the number of failures observed in an interval length $t$ and it only depends on the failure rate, which also depends on the state of the software. Due to the similarity of their probability
nature between time to failure and time to repair, Gokhale et al (2006) further argued that the repair time of software failures could also be modeled using NHCTMC and it is also shown that the time for maintenance activities behaves similar to time to failure and is successfully modeled in previous sections.

A NHCTMC can be uniquely characterized by its transition rate. We denote the process of software operation and maintenance as \( \{X_o(t)\} \) and \( \{X_m(t)\} \), respectively. Under the NHCTMC framework, the software failure rate and repair rate are respectively referred as \( \lambda(n,t) \) and \( \mu(n,t) \) where \( n \) stands for the total number of software failures encountered up to the present time \( t \). The mean value function of the operation and maintenance process, denoted as \( m_o(t) \) and \( m_r(t) \), can be obtained by integrating the failure rate and repair rate as shown below:

\[
m_o(t) = E[X_o(t)] = \int_0^t \lambda(n,s)ds\\
\]

\[
m_r(t) = E[X_m(t)] = \int_0^t \mu(n,s)ds\\
\]

With proper forms of failure rate and repair rate obtained with historical data, equations (3.12)&(3.13) provide a mathematical approach for analyzing software failure and repair events. If maintenance is called for every failure encounter, then with explicit failure rate and repair rate, we are able to predict the expected time to next failure as \( m_o^{-1}(n+1) - m_o^{-1}(n) \) and expected time for next repair as \( m_r^{-1}(n+1) - m_r^{-1}(n) \). The expected software availability for the next service cycle is:
Chapter III. Software Availability Modelling and Application Extensions

\[ E[A(n)] = \frac{m_o^{-1}(n+1) - m_o^{-1}(n)}{m_o^{-1}(n+1) - m_o^{-1}(n) + m_r^{-1}(n+1) - m_r^{-1}(n)} \]  \hspace{1cm} (3.14)

Equation (3.14) gives us a direct and explicit approach to estimate the expected software availability. However, one major issue in the modeling is how to estimate the failure and repair rate.

Different forms of failure rate and repair rate have been discussed by Gokhale et al. (2006). The parameters of failure rate and repair rate can be solely estimated based on sufficient historical data. In this research, we assume that both the failure and repair rate follow certain Non-Homogeneous Poisson Process (NHPP), which is independent of \( n \) and a special case of NHCTMC. Here we adopt the classical Geol-Okumoto model (Geol and Okumoto, 1979). With this Geol-Okumoto model, the failure and repair rate can be written as:

\[ \lambda(n,t) = a be^{-bt}, a, b > 0 \]  \hspace{1cm} (3.15)

\[ \mu(n,t) = \alpha \beta e^{-\beta t}, \alpha, \beta > 0 \]  \hspace{1cm} (3.16)

where \( a, b, \alpha, \beta \) are parameters to be estimated. These parameters also have physical meanings (Pham, 1999): \( a \) & \( \alpha \) stand for the total number of failures that can be eventually detected and corrected, respectively; \( b \) & \( \beta \) represent the efficiency of failure detection and correction. The mean value functions of the operation and maintenance process can be obtained as:
\begin{align}
m_o(t) &= a(1 - e^{-bt}) \\
m_r(t) &= \alpha(1 - e^{-\beta t})
\end{align}

Please note that we select the Geol-Okumoto model in this research merely as a matter of convenience. The proposed approach allows us to use other models as well. It is also important to verify the validity of the models used when our approach is applied with other real data sets. The parameters in the above equations can be estimated using either MLE or SLE (Xie, 1991; Lyu, 1996).

For the purpose of illustration, we will show how the model can be applied into practical use. Since data for software maintenance is hard to obtain and data from open source software is not suitable for such analysis, we adopt Gokhale’s approach (Gokhale et al, 2004; Gokhale et al, 2006) of analyzing software availability and a group of artificial data were generated from rate-based simulation. We will present a detailed theoretical support and simulation procedures for rate-based simulation later in Section 3.3.3 and in order to avoid redundancy, the simulation procedures are not described in this section. In this simulation, a software system is deployed and maintained. All-together 120 times of maintenance are recorded and the detailed time-line data is provided in the following table together with a process figure. In Table 3.4, Failure# counts the total number of software failure and maintenance encountered and \( U(t)/D(t) \) represents the total time spent in software operation and maintenance. In Figure 3.4, red line represents software operation while blue line represents software maintenance.
### Table 3-4 Simulation Data for Software Failure & Maintenance

<table>
<thead>
<tr>
<th>Failure#</th>
<th>$U(t)$</th>
<th>$D(t)$</th>
<th>Failure#</th>
<th>$U(t)$</th>
<th>$D(t)$</th>
<th>Failure#</th>
<th>$U(t)$</th>
<th>$D(t)$</th>
<th>Failure#</th>
<th>$U(t)$</th>
<th>$D(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>9</td>
<td>31</td>
<td>614</td>
<td>162</td>
<td>61</td>
<td>1553</td>
<td>346</td>
<td>91</td>
<td>2734</td>
<td>635</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>12</td>
<td>32</td>
<td>635</td>
<td>163</td>
<td>62</td>
<td>1592</td>
<td>347</td>
<td>92</td>
<td>2798</td>
<td>690</td>
</tr>
<tr>
<td>3</td>
<td>81</td>
<td>18</td>
<td>33</td>
<td>689</td>
<td>169</td>
<td>63</td>
<td>1728</td>
<td>351</td>
<td>93</td>
<td>2817</td>
<td>706</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
<td>22</td>
<td>34</td>
<td>743</td>
<td>176</td>
<td>64</td>
<td>1753</td>
<td>354</td>
<td>94</td>
<td>2823</td>
<td>717</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>30</td>
<td>35</td>
<td>787</td>
<td>177</td>
<td>65</td>
<td>1786</td>
<td>386</td>
<td>95</td>
<td>2949</td>
<td>732</td>
</tr>
<tr>
<td>6</td>
<td>109</td>
<td>34</td>
<td>36</td>
<td>828</td>
<td>185</td>
<td>66</td>
<td>1795</td>
<td>392</td>
<td>96</td>
<td>2971</td>
<td>740</td>
</tr>
<tr>
<td>7</td>
<td>115</td>
<td>55</td>
<td>37</td>
<td>830</td>
<td>194</td>
<td>67</td>
<td>1831</td>
<td>403</td>
<td>97</td>
<td>3034</td>
<td>747</td>
</tr>
<tr>
<td>8</td>
<td>123</td>
<td>60</td>
<td>38</td>
<td>896</td>
<td>197</td>
<td>68</td>
<td>1958</td>
<td>404</td>
<td>98</td>
<td>3099</td>
<td>767</td>
</tr>
<tr>
<td>9</td>
<td>132</td>
<td>70</td>
<td>39</td>
<td>899</td>
<td>199</td>
<td>69</td>
<td>1962</td>
<td>405</td>
<td>99</td>
<td>3124</td>
<td>781</td>
</tr>
<tr>
<td>10</td>
<td>152</td>
<td>71</td>
<td>40</td>
<td>901</td>
<td>206</td>
<td>70</td>
<td>1969</td>
<td>406</td>
<td>100</td>
<td>3178</td>
<td>804</td>
</tr>
<tr>
<td>11</td>
<td>191</td>
<td>74</td>
<td>41</td>
<td>908</td>
<td>219</td>
<td>71</td>
<td>1981</td>
<td>425</td>
<td>101</td>
<td>3463</td>
<td>818</td>
</tr>
<tr>
<td>12</td>
<td>194</td>
<td>75</td>
<td>42</td>
<td>922</td>
<td>221</td>
<td>72</td>
<td>2030</td>
<td>456</td>
<td>102</td>
<td>3487</td>
<td>820</td>
</tr>
<tr>
<td>13</td>
<td>206</td>
<td>76</td>
<td>43</td>
<td>927</td>
<td>222</td>
<td>73</td>
<td>2053</td>
<td>472</td>
<td>103</td>
<td>3533</td>
<td>826</td>
</tr>
<tr>
<td>14</td>
<td>233</td>
<td>89</td>
<td>44</td>
<td>962</td>
<td>225</td>
<td>74</td>
<td>2067</td>
<td>482</td>
<td>104</td>
<td>3545</td>
<td>835</td>
</tr>
<tr>
<td>15</td>
<td>271</td>
<td>92</td>
<td>45</td>
<td>981</td>
<td>242</td>
<td>75</td>
<td>2139</td>
<td>485</td>
<td>105</td>
<td>3553</td>
<td>864</td>
</tr>
<tr>
<td>16</td>
<td>275</td>
<td>102</td>
<td>46</td>
<td>1005</td>
<td>247</td>
<td>76</td>
<td>2142</td>
<td>486</td>
<td>106</td>
<td>3654</td>
<td>877</td>
</tr>
<tr>
<td>17</td>
<td>291</td>
<td>105</td>
<td>47</td>
<td>1049</td>
<td>249</td>
<td>77</td>
<td>2208</td>
<td>489</td>
<td>107</td>
<td>3761</td>
<td>936</td>
</tr>
<tr>
<td>18</td>
<td>378</td>
<td>107</td>
<td>48</td>
<td>1078</td>
<td>251</td>
<td>78</td>
<td>2210</td>
<td>493</td>
<td>108</td>
<td>3826</td>
<td>968</td>
</tr>
<tr>
<td>19</td>
<td>439</td>
<td>108</td>
<td>49</td>
<td>1088</td>
<td>260</td>
<td>79</td>
<td>2220</td>
<td>497</td>
<td>109</td>
<td>3883</td>
<td>1020</td>
</tr>
<tr>
<td>20</td>
<td>443</td>
<td>122</td>
<td>50</td>
<td>1103</td>
<td>277</td>
<td>80</td>
<td>2227</td>
<td>506</td>
<td>110</td>
<td>3896</td>
<td>1021</td>
</tr>
<tr>
<td>21</td>
<td>448</td>
<td>124</td>
<td>51</td>
<td>1171</td>
<td>278</td>
<td>81</td>
<td>2331</td>
<td>513</td>
<td>111</td>
<td>3921</td>
<td>1033</td>
</tr>
<tr>
<td>22</td>
<td>463</td>
<td>126</td>
<td>52</td>
<td>1196</td>
<td>284</td>
<td>82</td>
<td>2356</td>
<td>515</td>
<td>112</td>
<td>3999</td>
<td>1046</td>
</tr>
<tr>
<td>23</td>
<td>473</td>
<td>127</td>
<td>53</td>
<td>1267</td>
<td>296</td>
<td>83</td>
<td>2366</td>
<td>532</td>
<td>113</td>
<td>4089</td>
<td>1065</td>
</tr>
<tr>
<td>24</td>
<td>517</td>
<td>133</td>
<td>54</td>
<td>1280</td>
<td>306</td>
<td>84</td>
<td>2439</td>
<td>537</td>
<td>114</td>
<td>4181</td>
<td>1083</td>
</tr>
<tr>
<td>25</td>
<td>520</td>
<td>140</td>
<td>55</td>
<td>1320</td>
<td>320</td>
<td>85</td>
<td>2490</td>
<td>545</td>
<td>115</td>
<td>4194</td>
<td>1086</td>
</tr>
<tr>
<td>26</td>
<td>541</td>
<td>143</td>
<td>56</td>
<td>1331</td>
<td>324</td>
<td>86</td>
<td>2527</td>
<td>578</td>
<td>116</td>
<td>4375</td>
<td>1123</td>
</tr>
<tr>
<td>27</td>
<td>577</td>
<td>148</td>
<td>57</td>
<td>1334</td>
<td>329</td>
<td>87</td>
<td>2620</td>
<td>582</td>
<td>117</td>
<td>4395</td>
<td>1132</td>
</tr>
<tr>
<td>28</td>
<td>586</td>
<td>150</td>
<td>58</td>
<td>1431</td>
<td>333</td>
<td>88</td>
<td>2640</td>
<td>585</td>
<td>118</td>
<td>4419</td>
<td>1159</td>
</tr>
<tr>
<td>29</td>
<td>609</td>
<td>158</td>
<td>59</td>
<td>1491</td>
<td>335</td>
<td>89</td>
<td>2663</td>
<td>605</td>
<td>119</td>
<td>4558</td>
<td>1186</td>
</tr>
<tr>
<td>30</td>
<td>612</td>
<td>160</td>
<td>60</td>
<td>1504</td>
<td>337</td>
<td>90</td>
<td>2727</td>
<td>629</td>
<td>120</td>
<td>4662</td>
<td>1190</td>
</tr>
</tbody>
</table>
Figure 3-4 Plot of Simulated Software Process

We have used MLE to estimate the parameters and the results are shown in Table 3.5.

<table>
<thead>
<tr>
<th></th>
<th>Estimated</th>
<th>Upper Bound</th>
<th>Lower Bound</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>148.5</td>
<td>151.6</td>
<td>145.6</td>
<td></td>
</tr>
<tr>
<td>$b$</td>
<td>0.000344</td>
<td>0.000355</td>
<td>0.000332</td>
<td>2.779</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>139.1</td>
<td>142</td>
<td>136.2</td>
<td></td>
</tr>
<tr>
<td>$\beta$</td>
<td>0.00162</td>
<td>0.00168</td>
<td>0.00158</td>
<td>5.013</td>
</tr>
</tbody>
</table>
It can be concluded from the above results that the Geol-Okumoto model’s performance is acceptable and shows reasonable goodness-of-fit. The following figure represents the actual software operation and maintenance data versus the estimated fitting curve.

![Graph: Actual Data versus Estimated Curve](image-url)

**Figure 3-5 Actual Data versus Estimated Curve**
With the results obtained in Table 3.5, the estimated failure and repair rate could be obtained as:

\[
\lambda(n, t) = 148 \times 0.000344 \times e^{-0.000334t}
\]

\[
\mu(n, t) = 139 \times 0.00162 \times e^{-0.00162t}
\]

Then we are able to predict the availability of software systems:

<table>
<thead>
<tr>
<th>Expected</th>
<th>Upper Bound</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.77%</td>
<td>81.25%</td>
<td>69.19%</td>
</tr>
</tbody>
</table>

With the obtained failure and repair rates, the predicted expected availability of the software system is 75.77%, which is very low compared with industrial standards (usually 95% plus). In many cases, people require very high software availability. However, due to many external constraints, high software availability is costly and need to be carefully looked after. Much work has been done (Lee et al, 2005) in the purpose of achieving high software availability at affordable costs and we also plan to investigate this problem from the view point of software maintenance policy.
3.3 Software Maintenance Policy Considering Software Availability Constraints

In previous sections, the problems of software maintenance and software availability were investigated and modelled. Software maintenance affects software availability and in the meantime, people have requirements on software availability level, which affects software maintenance policy (Sahin and Zahedi, 2001). In this section, we will analyze software maintenance policy from a cost-effective criterion and show how our proposed software availability model can be applied in dealing with such problems.

3.3.1 Re-investigation on Software Maintenance and Software Maintenance Policy

Maintenance is a common practice in industry and many authors have studied the problem of optimal maintenance policy in industrial systems (Sahin and Zahedi, 2001). As a unique type of industrial system, the problem of software maintenance has also been studied by many authors (Ahmed, 2006; April et al, 2005), but most of these works focus on the qualitative analysis and few have ever quantitatively considered the impact of software maintenance on the software system’s service quality.

Software maintenance takes time and service quality of software system is degraded due to the unavailable time that is caused by software maintenance. The unavailable status of software servicing usually brings in severe consequences such as huge monetary loss. Usually a lot of effort is spent in maintenance with the aim of shortening the maintenance time (Jansen et al, 2006). Software failures usually require immediate attention (Tan and Mookerjee, 2005) and
it is desirable to request maintenance once a failure occurs. However, maintenance itself is also costly (Lientz and Swanson, 1981) and every time maintenance is performed, a setup cost incurs. This setup cost may include the cost for arranging repair facilities and personnel, as well as the cost of configuring the software systems for repair. For large software systems, setup cost for maintenance is sometimes so expensive that customers could not afford maintenance too often.

However, as software does not degrade as hardware does, a software outage could be temporarily solved by releasing all resources it is using and deleting all transactions it is executing and the most effective and efficient way of doing this is rolling back the corresponding transaction. This is called software rejuvenation (Avritzer et al, 2007; Huang et al, 1995; Jiang and Xu, 2007). Since there is software rejuvenation, software practitioners are able to record and roll back software transactions when a software failure occurs instead of calling for maintenance. Maintenance is not requested until certain threshold criteria are met. However, it is not easy to decide such threshold at which people call for maintenance. Most existing literatures which deals with maintenance policy avoids this problem by assuming that maintenance is driven by users’ request (Sahin and Zahedi, 2001) and explicit fault-removal activities are not covered. In most cases, existing methods on dealing with software maintenance policy only yield instructive proposals without convincing quantitative supports and this is also the main reason why many software companies are reluctant to use the existing proposed methods (Yu, 2006). One major advantage of quantitative analysis over qualitative analysis is that quantitative analysis yields repeatable numerical results, which can be used as decision making indicators.

Some preliminary assumptions are made before we further our investigation in software maintenance policy. The structure of software differs from system to system and the forms of software maintenance can also be different at different stages within the same software system. It
is hard to cover all the issues of software maintenance in one study. We aim at building a general model that can represent the most typical scenarios in industrial practice. In this thesis, only the corrective maintenance is considered. We further assert that software maintenance is carried out by maintenance personnel rather than online update or software patch installation. In continuation with our work done in previous sections, several assumptions are adopted in this section.

Assumptions:

1) Software failures are caused solely by software faults. Once a new failure occurs, it is recorded and then the corresponding software transaction is either rolled back or the software system goes under maintenance. Time for rolling back a software transaction is negligible and the software operation is assumed to be unaffected by rolling back the transactions.

2) Operation and maintenance environment remain the same all through the operational phase.

3) If a software maintenance schedule is initiated, the maintenance process removes all the recorded pending software failures. The maintenance activities will not introduce new software faults. In the mean while, operation is suspended until maintenance is done. The software operation and maintenance process both follow NHCTMC.

4) A software failure will not occur again if its corresponding software fault is removed. If a recorded software failure is recurring, it is not regarded as a new failure.
Most of the above assumptions are mathematical relaxation of the real situation except assumption 1. Actually, assumption 1 commonly exists in industrial software sets such as database software, manufacturing execution software systems and so on. In database software systems, there is a special SQL grammar called “ROLLBACK”, which is exactly designed to handle software failures as mentioned above. The above assumptions can be illustrated in the following figure. In Figure 3.6, the first maintenance is requested when the 3rd software failure was encountered and all the 3 software faults were corrected during the first maintenance interval. The upper line represents software operation activities while the lower line represents the software maintenance activities. $F_i, i = 1, 2, \ldots$ denotes the time at which the $i$th failure occurs and $R_i, i = 1, 2, \ldots$ denotes the time at which the $i$th fault is corrected. Comparing with Figure 3.2, the major difference is that in this section we explicitly state and allow multiple corrections in one maintenance interval, which is also the case in industrial practice (Kan, 2003; Darcy and Palmer, 2006; Eierman and Dishaw, 2007). Then the maintenance policy is quite apparent—to determine how many faults to be fixed during one maintenance interval.

![Figure 3-6 Demonstration of Operation and Maintenance of Software Systems](image-url)
3.3.2 Software Maintenance Policy Modeling with Consideration of Software Availability and Cost Constraints

The cost incurred during the software development and maintenance interests many researchers (Bhaskar and Kumar, 2006; Tan and Mookerjee, 2005; Zelkowitz et al, 1979). Maintenance cost commonly consists of three parts: setup cost, work cost and penalty cost (Feng et al, 2006) that are caused by unavailable time to users. Here we decompose the total cost of a maintenance schedule into the above-mentioned three main components.

(1) Setup cost: the cost incurred for preparing and arranging required resources to perform maintenance activities. In most cases, this type of cost is required and relatively stable when compared to other types of cost and is modeled as a known constant (Xie and Yang, 2003). In this research, the setup cost is regarded as a known constant which is denoted as $C_s$.

(2) Work cost: the cost incurred for attempting to remove the software faults within the software system. In most cases, this type of cost increases linearly with maintenance time and such phenomenon can simply be expressed as:

$$C_w = c_w t$$  \hspace{1cm} (3.19)

where $c_w$ is the unit time maintenance cost and $t$ is the maintenance time.

(3) Penalty cost: the cost incurred during the maintenance when the software system is unable to provide services. One apparent characteristic of the penalty cost is that it starts from zero but will increase very fast with time. However, the penalty cost cannot increase to infinity since decision makers would prefer to switch to a brand new system if the unavailable time is too
long. In this research, we propose a compound linear-exponential expression to model the unique properties of the penalty cost that are caused by unavailable time. The penalty cost is expressed as:

\[ C_p = c_p t(1 - \exp\{-\gamma t\}) \]  

(3.20)

where \( c_p \) is the cost coefficient, \( \gamma \) is a shape factor and \( t \) is the maintenance time. Equation (3.20) implies that the ultimate penalty cost first increases smoothly if the maintenance time is short but will increase dramatically if the maintenance takes longer time. If the system cannot be maintained any longer, then \( c_p \) stands for the ultimate unit time penalty cost.

With the above proposed costs, we are able to get the total cost expression. The total cost of a maintenance schedule can then be derived as:

\[ C_f(t) = C_s + C_w + C_p = C_s + c_w t + c_p t(1 - \exp\{-\gamma t\}) \]  

(3.21)

Equation (3.21) provides a mathematical approach for quantitative analysis of maintenance cost. \( C_s \), \( c_w \), \( c_p \) could be obtained from testing records or prior releases (Zhang and Pham, 2006; Kozlov et al, 2008). However, as our aim is to minimize the total maintenance cost all through the operational life cycle, equation (3.20) may not serve as a good decision factor of maintenance policy since it only represents the total cost of one maintenance schedule.

If the length of each maintenance schedule is reduced, the total cost of each schedule is reduced but the number of maintenance schedules increases, thus resulting high cost of
maintaining all the failures in the maintenance process. But if the length of each schedule is increased, the number of maintenance schedules is reduced but the total cost of each schedule increases, and it may also result in high cost of removing all the failures. To solve this paradox, we define unit time maintenance cost as a better metric. If the unit time maintenance cost is minimized in each maintenance schedule, the total maintenance cost of the whole operational life cycle can also be minimized. Thus the objective can be set as to minimize the expected cost per unit maintenance time.

However, time is not the only factor that software practitioners need to consider when scheduling a maintenance task. As one of the main purposes of software maintenance is to correct software faults, at least one software fault need to be removed during a software maintenance schedule otherwise such software maintenance is meaningless. Since the time required to maintain software failures vary each time so the time for maintenance itself is not a good maintenance objective. In industrial practice, software practitioners often set the number of software faults to be corrected in the next maintenance schedule as the maintenance objective. Denote the number of software faults to be corrected during a maintenance schedule as \( N \). It is obvious that \( N \) is inter-related with \( t \) and this figure is adopted as the decision variable in our research. Denote the expected unit time maintenance cost as \( E[C_A(t)] \) and then the optimality problem can be modeled as the following non-linear programming problem:
Chapter III. Software Availability Modelling and Application Extensions

Minimize: \[ E[C_A(t)] \]

subject to: \[ C_A(t) = \frac{C_T(t)}{t} \] \hspace{1cm} (3.22)

\[ C_T(t) = C_s + C_w + C_p = C_s + c_o t + c_p t(1 - \exp(-\gamma t)) \] \hspace{1cm} (3.23)

\[ X_m(t) = N \] \hspace{1cm} (3.24)

where: \( N \) is the decision variable and a positive integer

Equation (3.24) implies that the maintenance schedule ends when \( N \) faults have been removed. A closed-form solution of equation (3.22) alone can be mathematically derived by taking its first order partial derivative and setting it equal to zero:

\[ \frac{\partial C_A(t)}{\partial t} = 0 \] \hspace{1cm} (3.25)

Due to the existence of discrete constraint equation (3.24), equation (3.25) cannot serve directly as the final solution and we are not able to obtain a close-form solution for the above non-linear programming problem. However, due to the continuous property of equation (3.22), we can assert that the optimal solution lies either in \( \lfloor X_m(t) \rfloor \) or \( \lfloor X_m(t) \rfloor + 1 \), where \( \lfloor X_m(t) \rfloor \) stands for the largest integer that is no larger than \( X_m(t) \). If sufficient information is gained and cost model built, the problem can be easily solved numerically.
3.3.3 Validation of Policy Optimality—A Numerical Example

One major problem in quantitative analysis of software maintenance policy is the lack of supporting data. Software maintenance cost data, which is usually regarded as a part of business operation cost in practice, is often regarded as commercial confidential material and is hard to obtain. Many authors discussed maintenance policy from the qualitative point of view (Ahmed, 2006; Bhatt et al, 2006; Schneidewind, 2007) in order to avoid such problems. In this section, the example in Section 3.3.2 is extended to incorporate the cost issues. Since it is hard to get real industrial cost data, and as the purpose of illustration, we use some preset values of the proposed cost model as $C_s = 100, c_w = 10, c_p = 1000, \gamma = 0.5$ in the following analysis with the aim of proving our proposed model’s ability in solving the optimality problem. If field cost data is accessible, software practitioners can easily change the values and new results can be obtained with ease.

Theoretical support and simulation procedures

Since a closed form solution for the policy decision model is impossible to obtain, validation for optimality is a must. Simulation can take into account the operation process as well as the maintenance process in an integrated manner. Rate-based simulation is suitable for NHCTMC processes (Gokhale et al, 2004; Gokhale and Lyu, 2005) and it was used in previous sections for software availability modeling. It is also adopted here to simulate the operation and maintenance process.
A pure birth process is considered. If an event has not occurred by time \( t \), then for a pure birth process, the conditional probability that an event would occur in an infinitesimal interval \([t, t + dt]\) is given by \( \varphi(t)dt \), where \( \varphi(t) \) is called the event occurrence rate and for the operation and maintenance processes, it would correspondingly be the failure \( \lambda(n,t) \) and repair rate \( \mu(n,t) \), respectively.

The simulation procedure is described:

i) Proper value of parameters of failure rate, repair rate and cost model are chosen. Time increment \( dt \) is set to a reasonable value.

ii) \( N \) is assigned a value. Time is reset to 0.

iii) Time is increased by \( dt \). A random number \( x \) is generated and compared with \( \lambda(n,t)dt \).

If \( x < \lambda(n,t)dt \), a failure occurs and move to Step iv. Otherwise, repeat Step iii.

iv) Repeat Step iii \( N - 1 \) times.

v) Time is increased by \( dt \). A random number \( x \) is generated and compared with \( \mu(n,t)dt \).

If \( x < \mu(n,t)dt \), a fault is corrected and move on to Step vi. Otherwise, repeat Step v.

vi) Repeat Step v \( N - 1 \) times.

For a specific value of \( N \), the simulation procedure is run for a relatively large number of times and the average unit time cost is recorded and used as the expected value of unit time maintenance cost under the policy of \( N \). By comparing the results of different simulation results, the optimal maintenance policy can be obtained.
Simulation configuration and numerical results

As argued by Zhang and Pham (2006), we ought to have enough information about failure and repair rate before software deployment. Some authors (Koskinen et al, 2004; Kozlov et al, 2008) have also discussed the support for information needs of software maintainers. In this simulation, we extend the Apache example in Section 3.2.3 and assume that we gained enough information about the software system before deployment. The following configurations in Table 3.7 are used in the simulation. For the purpose of illustration and clarity, the units of all parameters are omitted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Notation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure rate</td>
<td>(\lambda(n,t))</td>
<td>(74.37 \times 0.5545 \times e^{-0.5545t})</td>
</tr>
<tr>
<td>Repair rate</td>
<td>(\mu(n,t))</td>
<td>(66.81 \times 0.2139 \times e^{-0.2139t})</td>
</tr>
<tr>
<td>Setup cost</td>
<td>(C_s)</td>
<td>100</td>
</tr>
<tr>
<td>Work cost</td>
<td>(c_w)</td>
<td>10</td>
</tr>
<tr>
<td>Penalty cost</td>
<td>(c_p)</td>
<td>1000</td>
</tr>
<tr>
<td>Penalty cost</td>
<td>(\gamma)</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Simulation programs are configured with the value of parameters provided above and the following results are obtained after ten thousand runs and analyses of the simulation results. Simulation results are provided in Table 3.8. In all, we obtained these results after 10k times of single simulation runs and the total estimated simulation time is about 10 hours. The standard deviation is less than 1% of the mean in each result and it is not shown here to avoid data cluster.
Table 3-8 Simulation Results

<table>
<thead>
<tr>
<th>N</th>
<th>( C_A(t) )</th>
<th>Expected maintenance time</th>
<th>N</th>
<th>( C_A(t) )</th>
<th>Expected maintenance time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1460.7</td>
<td>0.074</td>
<td>5</td>
<td>449.2</td>
<td>0.368</td>
</tr>
<tr>
<td>2</td>
<td>778.8</td>
<td>0.151</td>
<td>6</td>
<td>436.7</td>
<td>0.445</td>
</tr>
<tr>
<td>3</td>
<td>569.0</td>
<td>0.217</td>
<td>7</td>
<td>432.2</td>
<td>0.519</td>
</tr>
<tr>
<td>4</td>
<td>493.6</td>
<td>0.291</td>
<td>8</td>
<td>436.2</td>
<td>0.598</td>
</tr>
</tbody>
</table>

Conclusions can be made that the optimal maintenance policy is to conduct maintenance after 7 failures encountered. This would minimize the average unit time maintenance cost for each maintenance schedule. We also find out from our analysis that the setup cost is the main problem of high unit time maintenance cost in the beginning and then as the length of unavailable time increases, penalty cost dominates. So it is not economical to perform maintenance too often since the setup cost is too high, and it is neither economical to perform maintenance too late since otherwise the penalty cost would dominate.

However, from our results, the unit time maintenance cost of conducting maintenance for every 6 or 8 failures is not much over the optimal policy and the cost is subject to change since some conditions change regularly during the product life cycle. For example, cost of man power may increase; setup cost would go up due to the lack of necessary equipment, etc. Maintenance policies should stay stable even if such conditions change. Sensitivity analysis (Huang and Lyu, 2005b) is performed to check the stability of the policy’s optimality. We assume that the operation and maintenance process are stable and thus the main focus is on the cost parameters. An OFAT analysis is first conducted. The cost parameters in Table 3.7 are increased and
decreased by 10%, respectively and new results are calculated. The result of sensitivity analysis is shown in Table 3.9:

<table>
<thead>
<tr>
<th></th>
<th>+10%</th>
<th></th>
<th>-10%</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimal N</td>
<td>$C_A(t)$</td>
<td>Optimal N</td>
<td>$C_A(t)$</td>
</tr>
<tr>
<td>$C_s$</td>
<td>7</td>
<td>450.6</td>
<td>8</td>
<td>412.0</td>
</tr>
<tr>
<td>$c_w$</td>
<td>7</td>
<td>433.6</td>
<td>7</td>
<td>431.5</td>
</tr>
<tr>
<td>$c_p$</td>
<td>6</td>
<td>454.1</td>
<td>7</td>
<td>408.7</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>7</td>
<td>450.9</td>
<td>7</td>
<td>411.0</td>
</tr>
</tbody>
</table>

As can be concluded from the above table, the optimal maintenance policy stays stable when unit time work cost $c_w$ and penalty cost coefficient $\gamma$ changes within a certain interval. Although the expected unit time maintenance cost is changed in these two scenarios, the optimality of the original maintenance policy still holds. However, the results also show that the setup cost $C_s$ and the ultimate unit time penalty cost $c_p$ could affect the maintenance policy if they are changed. These comply with the fact that high setup cost will prolong the maintenance schedule and high penalty cost will shorten the maintenance schedule. We are also interested in the interaction between $C_s$ and $c_p$. A further 2x2 factorial analysis is conducted to check their level of interaction as in Table.3.10:
Chapter III. Software Availability Modelling and Application Extensions

Table 3-10 Interaction Analysis

<table>
<thead>
<tr>
<th>$C_s$</th>
<th>$c_p$</th>
<th>Optimal N</th>
<th>$C_A(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+10%</td>
<td>+10%</td>
<td>7</td>
<td>473.3</td>
</tr>
<tr>
<td>+10%</td>
<td>-10%</td>
<td>8</td>
<td>426.5</td>
</tr>
<tr>
<td>-10%</td>
<td>+10%</td>
<td>6</td>
<td>431.5</td>
</tr>
<tr>
<td>-10%</td>
<td>-10%</td>
<td>7</td>
<td>389.1</td>
</tr>
</tbody>
</table>

Effects of the setup cost and penalty cost can be obtained from the above table. Conclusions are made from Table 3.10 that in the presented scenario, the setup cost influences the maintenance policy more than the penalty cost. The situation occurs because in our example, the repair rate is relatively large and the maintenance does not take very long time. In all, the unit time work cost does not impact our decision making whereas increasing setup cost may prolong the maintenance cycle, and if the penalty cost can be reduced, a longer maintenance cycle is better than a shorter one.

3.4 Summary and Conclusion Remarks

Quality of software systems is of highly concern to both software vendors and software users, and many research efforts have been devoted into the field of software quality research, such as software reliability. However, as one major aspect of software quality, although the importance of software availability has been emphasized by many authors, there is still a lack of systematic analysis of software availability modeling and most of the existing work did not cover the origination problem that caused software unavailability—software maintenance. We are motivated to conduct a systematic analysis of software availability modeling, starting from the origination of the problem to the end-user applications of software availability modelling.
We first looked into the problem of software maintenance. As the main cause for software availability problems, software maintenance has been studied in many aspects and we aimed to build a proper model that can describe software maintenance activities. Maintenance effort was chosen as the break point and a software maintenance process model which was based on software maintenance efforts is proposed. The proposed model took explicit software maintenance activities into consideration and can provide important information such as software maintenance time with satisfactory confidence. Numerical examples also show that the proposed model can properly describe software maintenance process in an integrated manner.

With the obtained model that describes software maintenance process, we proposed a unified framework for estimating software availability. The software operation and maintenance processes are unified under a non-homogeneous continuous-time Markov chain and the software availability is obtained as the percentage of operational time in proportion to the total scheduled service time. Then we further investigated the problem of software maintenance policy. The problem of software maintenance policy has been covered by many authors, but most of these works are from a macro point of view and we thought it was worth considering specific constraints such as software availability into software maintenance policy making. The purpose of analyzing maintenance policy considering software availability was to achieve high software availability while keeping the maintenance cost at its minimal level. We carefully selected several criteria for determine the software maintenance policy and a cost model was built based on such criteria. By incorporating the cost model with the existing software availability model, we proposed a new approach to determine software maintenance policy. Numerical examples showed our proposed model’s ability in solving the optimality problem and the proposed model was also easy for practical use.
Chapter IV. Software Availability Assessment of N-version Programming Systems

Chapter 4 Software Availability Assessment of N-version Programming Systems

In previous chapter, a general model of software availability was proposed for estimating software availability metrics and the proposed model could be applied in practice. However, as a specified model for traditional single version software system it is, the proposed model cannot be directly applied for assessing the software availability of certain special types of software systems, such as the fault-tolerant software systems. Complex software systems like the fault-tolerant software systems are carefully designed based on redundancy mechanism and traditional analysis approach is not efficient in dealing with such systems. The work done in this area regarding software quality analysis is limited (Avizeinis, 1985) and few have ever considered to analyze fault-tolerant mechanism’s impact on software availability. To alleviate these drawbacks, we are motivated to propose a framework for estimating the software availability of complex fault-tolerant software systems. Numerical validation shows that the proposed methods can successfully estimate software availability properties of fault-tolerant software systems and can serve as indicators for further decision making.

4.1 Introduction

Fault-tolerance of software systems refers to the property that enables a software system to continue operating properly in the event of the failures due to one or more faults within some of its components. If its operating quality decreases at all, the decrease is proportional to the
severity of the failure, as compared to a naively-designed system in which even a small failure can cause total breakdown (Avizienis, 1985; Avizienis, 1995). Software fault-tolerance is also called graceful software degradation by some authors (Avizienis, 1995) and fault-tolerance is particularly sought-after in high-availability or life-critical systems (Pham, 1992).

Fault-tolerant software is a direct consequence of the balance between requirement of high-quality software and high cost of building such systems. Dysfunctions or malfunctions that are caused by software failures in complex systems bring a lot of trouble to system users, and any dysfunction or malfunction in a safety critical system may lead to direct disastrous consequences (Pham, 2003). Since testing may not eliminate all software faults (Xie, 1991; Lyu, 1996) and intensive thorough software testing is very expensive (Xie and Yang, 2003), an affordable method to improve the quality of complex systems is required. Software fault tolerance is such a method (Aghdaie, 2009). There are primarily two available types of fault-tolerance techniques in use. One is the recovery block technique (Randell, 1975; Berman and Kumar, 1999) and the other is the N-version programming technique (Chen and Avizienis, 1978; Avizienis, 1985), which is more popular (Avizienis, 1995; Yamachi et al, 2006). Although different in forms and mechanisms, both recovery block and N-version programming techniques are based on redundancy and the mathematical representation of these two can be unified (Avizienis, 1995). In this thesis, only the N-version programming is covered as it is easier to design and deploy and most of the fault-tolerant software systems in practical use adopt N-version programming (Avizienis, 1995).

N-version programming, also known as multi-version programming, is a method or process in software engineering where multiple functionally equivalent programs are independently generated from the same initial specifications. The authors (Chen and Avizienis,
1978) who first introduced N-version programming argued that "independence of programming efforts will greatly reduce the probability of identical software faults occurring in two or more versions of the program".

The N-version programming technique is mainly based on redundancy (Taboada et al, 2008; Maciejewski and Caban, 2008; Ouzineb et al, 2008; Alzamil, 2008) and it involves the parallel execution of several independently developed but functionally equivalent versions of a single program. Each single version behaves similarly as ordinary software program, but the N-versions as a whole have better performance in many aspects.

In the early stage of N-version programming research, authors mainly focused on the different modeling of N-version programming (Littlewood and Miller, 1989; Nicola and Goyal, 1990). These works discussed the possibility of simultaneous failures among different versions and this topic was later continued with further research as failure correlation in N-version programming software systems (Gutjahr, 2001; Dai et al, 2004; Levitin and Xie, 2006). As the idea of N-version programming was first inspired from enhancing software reliability engineering (Avizienis, 1985), most of the reported literatures discussed N-version programming technique’s ability in enhancing software reliability (Dugan and Lyu, 1994; Scott et al, 1987) and in the recent years, there is more and more literature which deals with N-version programming problems under software reliability constraints. Some authors have their focus on the cost (Bhaskar and Kumar, 2006) and optimal structure of N-version programming (Dai et al, 2004; Levitin, 2004; Levitin and Xie, 2006; Levitin, 2005; Yamachi et al, 2006) while some others take the imperfect debugging scenario into consideration (Chatterjee et al, 2004).

However, although plenty of efforts have been put into the research on N-version programming, few authors have ever emphasized N-version programming technique’s impact on
software availability. One major reason for such situation is that software reliability analysis was emphasized so much in these works that software availability was regarded as a by-product of software reliability analysis. In order to alleviate such problems, we are motivated to conduct a thorough study on N-version programming systems to check its impact on software availability. In this chapter, we propose a novel Markov chain-based software availability model of N-version programming system. While the research in this area is still inadequate, our proposed model may help readers in understanding the software availability issue step by step with better estimation accuracy. Under this model, behavior of each sub-version program is explicitly considered and analyzed. This model makes the software process more apparent and understandable for users and may serve as a decision-making tool for software practitioners who seek building robust software systems with high availability and budget restrictions.

The remainder of this chapter is organized as follows: Section 4.2 first investigates the structure of N-version programming system. The behavior of each version is explicitly modeled using a Makovian approach and then the software availability model is derived. Section 4.3 presents several numerical examples to show N-version programming’s impact on software availability with comparisons to traditional single version software systems. In Section 4.4, the problem of constructing robust N-version programming software systems is covered and analyzed with quantitative results.

4.2 Software Availability Modeling of N-version Programming Systems

N-version programming receives much attention of software practitioners, especially of those from safety-critical industries. The design and development of N-version programming
software systems are fundamentally different from traditional single-versioned software systems. The structural difference between N-version programming systems and traditional systems results in different performance in software reliability and availability. In this section, the structural architecture of N-version programming systems is first investigated. Since the status of the whole system depends on its sub-versions, we propose a Markov chain to model the behaviors of each individual sub-version. Under the assumption of none failure correlation, the behaviors of each individual sub-version are integrated and the software availability model is obtained.

4.2.1 Structure Decomposition of N-version Programming Systems

N-version programming mechanism is basically based on the redundancy created by functionally equivalent but independently developed software modules that are called versions. It has received considerable attention from software practitioners (Avizeinis, 1985; Chatterjee et al, 2004; Wattanapongsorn and Coit, 2007), especially in safety critical systems like airport control system and missile navigation system. Though suffering criticism such as requiring trade-off between additional computing resources and speed performance (Gutjar and Uchida, 2002; Wattanapongsakorn and Levitan, 2004), N-version programming outperforms traditional software design methods from the view of reliability consideration (Teng and Pham, 2002). And with the fast advancing hardware technology, computing resources are no longer the key constraint for software performance and N-version programming is becoming more and more widely applied in different software systems (Sha, 2001) because of the increasing demand of high-performance software.
Just as its name indicates, an N-version programming software system usually consists of two or more structurally different but functionally equivalent versions of execution paths (Avizeinis, 1985). When data is input, it is processed separately and simultaneously by different versions. Versions work separately and collaborate under a voting mechanism. Output by each individual version is compared and the most suitable ones are selected by a voting algorithm as the final output so that failures within minority versions may not affect the final output. Searching for a proper voting algorithm in each specific N-version programming systems are vital to its success and it is hard to find a proper one. Many authors have been working on this topic for a long time (Chen and Avizeinis, 1978) with fruitful and mature investigations but the majority voting mechanism is still the most popular one. In the majority voting mechanism, the system simply collects the simple majority as the final output of the whole system. And in this chapter, we adopt the majority voting mechanism. In all, an N-version programming software system is a software system that utilizes redundancy to reduce the probability of encountering failure. Much recent work has been done on studying redundancy in hardware systems (Taboada et al., 2008; Maciejewski and Caban, 2008; Ouzineb et al., 2008), however, redundancy in fault-tolerant software is somehow different and it is a repairable system (Zhang, 2008). A simple comparison between traditional software and N-version programming software is illustrated in the following figure.
Figure 4.1 Traditional Software vs. N-version Programming Software

In Figure 4.1, it is assumed that the input data will be serially processed by 4 different modules before the final output is generated. In the traditional single-versioned software, the input data is processed only once. In the N-version software, the input data was simultaneously processed by \( n \) different but functionally equivalent sub-versions and the output of each version is compared before the final output is generated.

Ideally speaking, different versions in the N-version software system should be independent so as to reduce the possibility of failure triggered by the same input. Such input dependency between different versions is called failure correlation by some authors (Levitin and Xie, 2006; Dai et al, 2004; Xie et al, 2005). However, this ideal case cannot be guaranteed and different versions tend to be correlated to some extent, though slightly (Teng and Pham, 2002). Some authors (Pham, 1992; Knight and Leveson, 1986) experimentally revealed that though different versions were developed independently by different teams, they do not necessarily fail independently. Laprie et al (1990) classified faults among different version as either
independent faults or related common faults and identified that common faults usually took only a very small portion in all the faults. In this section, we begin our investigation from the starting point and only the ideal case that all versions work and cooperate independently is taken into consideration.

4.2.2 Proposed Model of Software Availability of N-version Programming Systems

Software availability of single version software has been studied by some researchers. However, few of them have ever realized the potential of N-version programming systems in this aspect. With its competency in enhancing software quality in many aspects, it is reasonable to assume N-version programming would have a positive impact on software availability.

In the previous chapter, we proposed a software availability model based on non-homogeneous continuous time Markov chain. Numerical validations showed that the proposed model worked well for traditional single-versioned software; however, it is hard to apply the proposed model directly to N-version programming system. On one hand, the failures of N-version programming are harder to trace than that of the single-versioned software systems since minor failures are “absorbed” by the fault-tolerant mechanism. On the other hand, failures in each sub-version of N-version programming system are hard to be described using one mean value function. A new approach is needed for the software availability investigations on N-version programming systems. We make several preliminary assumptions in order to simplify the complex system before we build our software availability model.
Preliminaries

N-version programming techniques are often applied to large software systems, in which statistical deduction is meaningful. A typical N-version programming system which consists of $n$ sub-versions is considered. The target system is based on the majority voting mechanism. We focus on the failure events of each single sub-versions and the N-version programming system as a whole. The following four assumptions are presumed

Assumptions:

1. An N-version software system consists of several different and independent sub versions of execution paths. Data is processed simultaneously on all the execution paths.
2. Each of the sub versions of execution paths has a number of initial faults, which can trigger failures in each corresponding path in operation. Faults in different versions are not correlated.
3. When a failure occurs in an execution path, this version of execution path stops working and will be restored. Failure restoration takes time and efforts. Work in other versions is not interfered and restored faults do not occur again.
4. When more than half of all the sub versions of paths fail, the software system is considered as failed.

Mathematical Formulation of Software Availability Model of N-version Programming System
Chapter IV. Software Availability Assessment of N-version Programming Systems

For an N-version programming system which is described above, all *n* sub-versions are functionally equivalent, so it is reasonable to believe that all *n* versions should behave similarly. In this case, we are able to analyze the failure and restoration behaviors of an individual sub-version first and then the behaviors of other versions can be concluded.

The state of a single sub-version *i* can be denoted using a duplet \((j(t), w)\) where \(j(t) = 0, 1, ..., N_i\) is the total number of failures encountered in version *i* up to time *t* and \(w = \begin{cases} 0 \\ 1 \end{cases}\) indicates this version of execution path is working (indicated by 0) or non-working (indicated by 1) in restoration. For a single sub-version, it alternates between up and down stages. State transition diagram of a single version *i* can be illustrated using the graph below:

![State Transition Diagram of Sub-version i](image)

**Figure 4-2 State Transition Diagram of Sub-version i**

A stochastic process \(\{X_i(t), t \geq 0\}\) and \(X_i(t) = (j(t), w)\) is considered. The process that is presented in Figure 4.2 can be modeled as a pure birth Markov chain. For version *i*, the time to next failure is exponentially distributed with mean time \(\frac{1}{\lambda_{i,j}}\) while the time to next restoration is
exponentially distributed with mean time $\frac{1}{\mu_{i,j}}$. Restorations can be done either by professionals or by self-checking mechanism.

Further assumption is that no failures would occur in any version at the beginning of operation, which means the target N-version programming is in a fully functioning state at the beginning. Note that two different definitions of software availability are covered and unified in the beginning of the previous chapter. Since the mean value function is hard to obtain for N-version programming systems, in this chapter we adopt the software availability definition as the probability at a given time that the software system is working and accessible. Denote $A(t)$ as the software availability at time $t$. Then according to our assumptions, the software availability of an N-version software at time $t$ is the probability that no less than half of its sub-versions are still functioning. Since $w_i$ can be used to indicate that sub-version is working(0) or non-working(1), by summing up $w_i$ we can have a rough picture of the status of the entire system and the software availability can be expressed as:

$$A(t) = \Pr\left\{ \sum_{i=1}^{n} w_i \leq \left\lfloor \frac{n}{2} \right\rfloor \mid X_i(t) = (j_i, w_i), i = 1, 2, \ldots, n \right\}$$  \hspace{1cm} (4.1)

where $\left\lfloor \frac{n}{2} \right\rfloor$ is the floor function which returns the greatest integer that is no larger than $\frac{n}{2}$.

This availability $A(t)$ means the software system is operational at time $t$, given that the system was operational at time 0. Furthermore, the average software availability over the period $(0, t]$ can be calculated as
Chapter IV. Software Availability Assessment of N-version Programming Systems

\[ \bar{A}(t) = \frac{1}{t} \int_0^t A(s)ds \]  

(4.2)

and the average software availability represents the average proportion of operating time during the time interval \((0,t]\), given that the software is operational at time 0. In most cases, this average figure is more meaningful and more widely used. However, equation (4.1) and equation (4.2) only give a conceptual demonstration of how software availability can be obtained and further derivation is required if quantitative measurement of software availability using explicit expressions is needed.

Since each sub-version works independently, behaviors of a single version can be first analyzed. The probability that version \(i\) will be in which certain state after it has elapsed time \(t\) is of the research interest. Denote \(p^i_{(j,0),(k,0)}(t)\) as the probability that version \(i\) will be in the working state at time \(t\) after \(k\) faults are to be removed, given that the software is currently in the working state at time 0 and \(j\) faults have been removed previously, can be written as

\[ p^i_{(j,0),(k,0)}(t) = \text{Pr}\{X_i(t) = (k,0) \mid X_i(0) = (j,0)\}, j \leq k \]  

(4.3)

for \(j = 0,1,2,\ldots,k\) and \(k = 0,1,2,\ldots,N_i\). The availability of this sub version \(i\) at time \(t\) is the sum of the probability that this version \(i\) has experience several failure and restorations but is still in “up” state at \(t\) and can be calculated as
Chapter IV. Software Availability Assessment of N-version Programming Systems

\[ A^i(t) = \sum_{j=0}^{N_i} p_{(0,0,j,0)}^i(t) \] (4.4)

According to the assumptions stated above, the entire software system is available only when no less than half of its subversions are functioning. If the availability of a single version can be obtained, the availability of the entire system can also be derived directly.

**Proposition:**

\[ p_{(0,0,N_{i,0})}^i(t) = \{ \prod_{l=0}^{k-1} \lambda_{i,l} \mu_{i,l} \} \sum_{j=0}^{k} \left[ A_j + \mu_{i,k} \right] S_A(j,k) \exp(A_j t) + \frac{B_j + \mu_{i,k}}{S_B(j,k)} \exp(B_j t) \}, k < N_i \] (4.5)

\[ p_{(0,0,N_{i,0})}^i(t) = 1 + \{ \prod_{l=0}^{k-1} \lambda_{i,l} \mu_{i,l} \} \sum_{j=0}^{N_{i-1}} \left[ \frac{\exp(A_j t)}{A_j S_A(j,l)} + \frac{\exp(B_j t)}{B_j S_B(j,l)} \right] \] (4.6)

where

\[
\begin{align*}
S_A(j,k) &= \prod_{l=0}^{k} (A_j - B_l) \prod_{m=0}^{k} (A_j - A_m) \\
S_B(j,k) &= \prod_{l=0}^{k} (B_j - B_l) \prod_{m=0}^{k} (B_j - A_m)
\end{align*}
\] (4.7)

and \( A_j \) and \( B_j \) are two roots of the quadratic equation \( x^2 + (\lambda_{i,j} + \mu_{i,j})x + \lambda_{i,j} \mu_{i,j} = 0 \).
Chapter IV. Software Availability Assessment of N-version Programming Systems

The general expression of availability of the N-version software system can be derived by adapting the working state probability

\[ A(t) = \sum_{i=\lceil n/2 \rceil}^{n} \left( \prod_{j=1}^{i} \left[ \zeta_j \right] \right) \]

(4.8)

where \( \zeta_{j1} \) and \( \zeta_{j2} \) are defined as:

\[ \zeta_{j1} = \left[ 1 - \sum_{k=0}^{N_i} p_{(0,0,k,0)}^{(i+1,j)}(t) \right] \cdots \left[ 1 - \sum_{k=0}^{N_i} p_{(0,0,k,0)}^{(0,j)}(t) \right] \]

\[ \zeta_{j2} = \sum_{k=0}^{N_i} p_{(0,0,k,0)}^{(1,j)}(t) \sum_{k=0}^{N_i} p_{(0,0,k,0)}^{(2,j)}(t) \cdots \sum_{k=0}^{N_i} p_{(0,0,k,0)}^{(j,j)}(t) \]

where \( p^{1j} p^{2j} \cdots p^{jj} \) denotes a unique combination of \( i \) different versions out of \( n \). Substituting the working state probability by equation (4.4) and equation (4.5) into the above expression, the explicit expression for software availability of N-version programming systems can be obtained.

**Proof Begins:**

Let \( F_{i,\beta}(t) \) be the one-step transition probability that the process of version \( i \) at present time in state \( \alpha \) will be in state \( \beta \) after elapsed time \( t \). The transition probabilities can be written as:

\[ F_{(j,0)(j+1)}(t) = 1 - e^{-\lambda_i \cdot t}, \quad j < N_i \]  

(P1)

\[ F_{(j,0)(j+1)}(t) = 1 - e^{-\mu_i \cdot t}, \quad j < N_i \]  

(P2)

Suppose version \( i \) is currently in working state and \( j \) faults have already been encountered and restored. Let \( T_{(j,0)}^{(i)} \) represent the first passage time of the version \( i \) to the \( k \)th fault removal,
where \( j < k \leq N_i \) and \( N_i \) is the number of initial faults of version \( i \). The distribution function of \( T_{(j,0)(k,0)}^i \), denoted by \( G_{(j,0)(k,0)}^i(t) \), can be expressed as:

\[
G_{(j,0)(k,0)}^i(t) = F_{(j,0)(k,0)}^i(t) * F_{(j,1)(k,0)}^i(t) * G_{(j+1,0)(k,0)}^i(t)
\]  \hspace{1cm} \text{(P3)}

for \( j = 0,1,2,...,k-1; k = 1,2,3,...,N_i \), and * symbolizes the Stieltjes convolution. By taking the Laplace-Stieltjes (L-S) transform of both sides of (P3), a new equation could be obtained:

\[
\tilde{G}_{(j,0)(k,0)}^i(s) = \tilde{F}_{(j,0)(k,1)}^i(s)\tilde{F}_{(j,1)(k,0)}^i(s)\tilde{G}_{(j+1,0)(k,0)}^i(s)
\]  \hspace{1cm} \text{(P4)}

where \( \sim \) denotes the L-S transform. Using (P1) and (P2), one can further obtain:

\[
\tilde{F}_{(j,0)(k,1)}^i(s) = \frac{\lambda_{i,j}}{s + \lambda_{i,j}}
\]  \hspace{1cm} \text{(P5)}

\[
\tilde{F}_{(j,1)(k,0)}^i(s) = \frac{\mu_{i,j}}{s + \mu_{i,j}}
\]  \hspace{1cm} \text{(P6)}

Substituting (P5) and (P6) into (P4), one can get

\[
\tilde{G}_{(j,0)(k,0)}^i(s) = \frac{\lambda_{i,j}\mu_{i,j}}{(s + \lambda_{i,j})(s + \mu_{i,j})}\tilde{G}_{(j+1,0)(k,0)}^i(s)
\]  \hspace{1cm} \text{(P7)}

An explicit solution for general \( j \& k \) of (A7) is hard to obtain. However, the time period that interests us is from the very beginning of software operation till its failure time. In other words, the special case with \( j=0 \) is our research focus. According to the property of L-S transform, \( \tilde{G}_{(k,0)(k,0)}^i(s) = 1 \) for all \( k \geq 0 \). Starting from \( j=0 \), by recursively replacing (P7) with (P5) and (P6), then (P7) could be solved with boundary condition \( \tilde{G}_{(k,0)(k,0)}^i(s) = 1 \). The solution is

\[
\tilde{G}_{(0,0)(k,0)}^i(s) = \frac{\lambda_{i,0}\mu_{i,0}}{(s + \lambda_{i,0})(s + \mu_{i,0})}\tilde{G}_{(1,0)(k,0)}^i(s) = \frac{\lambda_{i,0}\mu_{i,0}\lambda_{i,1}\mu_{i,1}}{(s + \lambda_{i,0})(s + \mu_{i,0})(s + \lambda_{i,1})(s + \mu_{i,1})}\tilde{G}_{(2,0)(k,0)}^i(s)
\]

\[
= \cdots = \prod_{j=0}^{k-1} \frac{\lambda_{i,j}\mu_{i,j}}{(s + \lambda_{i,j})(s + \mu_{i,j})}\tilde{G}_{(k,0)(k,0)}^i(s) = \prod_{j=0}^{k-1} \frac{\lambda_{i,j}\mu_{i,j}}{(s + \lambda_{i,j})(s + \mu_{i,j})}
\]  \hspace{1cm} \text{(P8)}

\[96\]
For the probability \( p_i^{(j,0)(k,0)}(t) \), it is very clear that \( p_i^{(j,0)(k,0)}(t) = \exp\{-\lambda_{ij}t\} \) and \( p_i^{(N,0)(N,0)}(t) = 1 \). Utilizing the first passage time obtained above

\[
p_i^{(j,0)(k,0)}(t) = G_i^{(j,0)(k,0)} \neq p_i^{(k,0)(k,0)}(t) \tag{P9}
\]

By taking the L-S transform of both sides of (P9), one could obtain

\[
\tilde{p}_i^{(j,0)(k,0)}(s) = G_i^{(j,0)(k,0)}(s) \tilde{p}_i^{(k,0)(k,0)}(s) \tag{P10}
\]

Similarly, the solution of a special case with \( j=0 \) is the objective. Since

\[
\tilde{p}_i^{(k,0)(k,0)}(s) = \frac{1}{s + \lambda_{ik}}, \quad \text{and} \quad \tilde{p}_i^{(N,0)(N,0)}(s) = \frac{1}{s},
\]

substituting (P8) into (P10), one can obtain

\[
\tilde{p}_i^{(0,0)(k,0)}(s) = \frac{1}{s + \lambda_{ik}} \prod_{j=0}^{k-1} \frac{\lambda_{ij}\mu_{ij}}{(s + \lambda_{ij})(s + \mu_{ij})}, k < N_i
\]

\[
\tilde{p}_i^{(0,0)(N,0)}(s) = \frac{1}{s} \prod_{j=0}^{N-1} \frac{\lambda_{ij}\mu_{ij}}{(s + \lambda_{ij})(s + \mu_{ij})}
\]

(P11) can be further formulated as

\[
\tilde{p}_i^{(0,0)(k,0)}(s) = \prod_{j=0}^{k-1} (\lambda_{ij}\mu_{ij}) \left\{ \prod_{j=0}^{k-1} \frac{1}{(s - A_j)(s - B_j)} \right\} (s + \mu_{ik}), k < N_i
\]

\[
\tilde{p}_i^{(0,0)(N,0)}(s) = \prod_{j=0}^{N-1} (\lambda_{ij}\mu_{ij}) \left\{ \frac{1}{s} \prod_{j=0}^{N-1} \frac{1}{(s - A_j)(s - B_j)} \right\}
\]

where \( \{A_j\} \) are two roots of the quadratic equation \( x^2 + (\lambda_{ij} + \mu_{ij})x + \lambda_{ij}\mu_{ij} = 0 \). By inverting the above equation, one can get

\[
p_i^{(0,0)(k,0)}(t) = \{ \prod_{j=0}^{k-1} \lambda_{ij}\mu_{ij} \} \sum_{j=0}^{k} \frac{A_j + \mu_{ik}}{S_A(j,k)} \exp(A_jt) + \frac{B_j + \mu_{ik}}{S_B(j,k)} \exp(B_jt) \}, k < N_i
\]
Chapter IV. Software Availability Assessment of N-version Programming Systems

\[
P^i_{(0,0)(N,0)}(t) = 1 + \left\{ \prod_{i=0}^{N-1} \lambda_{i,j} \mu_{i,j} \right\} \sum_{j=0}^{N-1} \left\{ \frac{\exp(A_j t)}{A_j S_A(j,1)} + \frac{\exp(B_j t)}{B_j S_B(j,1)} \right\},
\]

where

\[
S_A(j,k) = \left\{ \prod_{i=0}^{k} (A_j - B_i) \right\} \left\{ \prod_{m=0}^{k} (A_j - A_m) \right\},
\]

\[
S_B(j,k) = \left\{ \prod_{i=0}^{k} (B_j - B_i) \right\} \left\{ \prod_{m=0}^{k} (B_j - A_m) \right\}
\]

Proof Ends

4.3 Impact of N-version Programming on Software Availability

In the previous section, the software availability model for N-version programming is explicitly derived. However, the model itself can only be used for assessing software availability and we still need to investigate how N-version programming can affect the availability performance of software systems. In this section, a 2-out-of-3 N-version programming software system is considered. Although demonstrative, this section clearly reveals the potential of N-version programming in enhancing software availability performance and the conclusions can easily be generalized to any general \( k \)-out-of-\( n \) N-version programming systems. Based on our analysis, we further investigate the problem of optimal structure of N-version software systems and provide specific suggestion for practical use.

4.3.1 Failure and restoration rates for different versions of software
Empirical Analysis of Software Operation and Restoration Time

The operation and restoration time for software systems is an important metric for software users. Many authors have reported on this issue. According to Tamai and Torimitsu’s survey (1992), the average lifetime for mainframe-level business software systems is about 9 years and this figure increases as the size (usually measured in kilo lines of code) of software increase while only minor maintenance work is accumulated until a thorough system upgrade. Burch and Kungs (1997) also studied the repair work of a business software system which lasted 67 months. An average of only 4.78 repairs per month was conducted during the entire 67 months and a clear statistical decreasing trend of software repairs is also shown in their paper. Jacoby and Tohma (1991) further showed that the average restoration time should take less than 30% of the total time in order to be sufficient to correct all software faults. This finding provides a rough ratio between software operation and restoration time. Although the above authors did not provide explicit data on software restoration time, it is reasonable to assume that the time spent on software restoration is much less than software operation time according to their findings.

It is hard to get industrial field data of software operation and restoration for practical analysis, but simulation analysis based on empirical basis is still meaningful. Appropriate failure and restoration rates can be used to represent software operation and restoration time, based on which software availability can be assessed. In reality, it is possible for a real software process to differ from the simulated process, but this problem can be solved by replacing failure and restoration rates which better model the process. Hence, the simulation analysis can be rationally applied in practice with minor changes.
Mathematical Representation of Software Failure and Restoration Rates

Generally speaking, failure rate of the software system would gradually decrease as more faults are removed with time, while the restoration rate would also decrease with time since the complexity (Nakagawa and Takenaka, 1991) of faults encountered later is increased.

Moranda (1979) proposed a geometric model from the viewpoint that software reliability depends on debugging efforts and earlier debugging activities have larger impact on software reliability growth than later ones. This model has been discussed and adopted by many researchers and is still popular with software practitioners (Dick et al., 2007) and we also use Moranda’s model to describe the software failure-occurrence phenomenon, i.e., when $j$ faults have been corrected, the hazard rate is given by

$$\lambda_{i,j} = D_i k_i^j, 0 < k_i \leq 1, j \geq 0$$  \hfill (4.9)

where $D_i$ is the initial hazard rate in version $i$ and $k_i$ is a proportionality factor. For the restoration rate, we assume it is similar to hazard rate and takes the form of (Takuno and Yamada, 2003)

$$\mu_{i,j} = E_i r_i^j, 0 < r_i \leq 1, j \geq 0$$  \hfill (4.10)

where $E_i$ is the initial failure restoration rate of version $i$ and $r_i$ is a proportional factor. By substituting equation (4.9) and equation (4.10) into equation (4.8), an explicit software
availability expression is obtained. Due to the page size limit, the explicit expressions will not be presented here.

Note that we choose Moranda’s model to represent both failure and restoration rate merely as a matter of convenience. Our approach apparently allows us to use other models. For real data, it is also important to test the validity of the used model.

In real life scenarios, plenty of approaches have been proposed for estimating the parameters in equation (4.9) & equation (4.10) with the historical data that was recorded during the development phase (Xie, 1991; Lyu, 1996). Other useful figures such as the number of remaining faults in the software can also be obtained using these approaches.

4.3.2 Impact of N-version Programming on Software Availability: A Simulation Approach

Interactions among sub-versions of an N-version programming system should greatly impact greatly on the system’s availability. Due to the uniqueness of different sub-versions, availability performance of the entire N-version system could vary if different sub-versions are chosen. In this section, we aim at finding out the possible availability behaviors of a 2-out-of-3 N-version programming system by considering different sub-version contents. A 2-out-of-3 N-version programming system will not fail if no more than 2 sub-versions fail simultaneously. The reason that we choose the 2-out-of-3 N-version programming system is because it is simple for illustration as well as commonly used in industrial cases (Avizienis, 1995). In the mean time, the results can also easily be expanded to any general k-out-of-n N-version programming systems.

N-version programming systems are usually applied in safety-critical projects and the operation/maintenance data are usually kept as confidential materials and it is almost impossible
to get field data from commercial companies. On the other hand, the target uses of N-version programming software systems is limited when compared to traditional software systems, and this reason also confines the development of N-version software in open source software community, which requires a large number of participants.

In order to avoid the problems above, a simulation approach is adopted in this research. According to the probabilistic nature of its behavior, we also adopt the rate-based simulation in this section. Three different software versions are simulated. It is assumed that they are developed with different fault content. Each version is capable of performing required tasks independently. Different software versions as well as their collaboration are simulated using different version parameters (failure/restoration rates) that are stated above and the simulation program is developed with Java. For the purpose of illustration, all parameters are preset with reasonable value. For real industrial applications, these parameters can be replaced by field data.

**Scenario One**

An ideal case is first considered in this scenario and we would like to check the ability of N-version in enhancing software availability in positive environments. Suppose three well-developed sub-version programs are provided. For the purpose of comparison, we assume all different sub-versions are with the same initial number of faults to avoid quantitative dominance of faults (Huang and Lyu, 2005). Details of each version are provided in the following table.

<table>
<thead>
<tr>
<th>Table 4-1 Details of 3 Versions in Scenario One</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
</tr>
</tbody>
</table>
Chapter IV. Software Availability Assessment of N-version Programming Systems

<table>
<thead>
<tr>
<th>Version</th>
<th>( \text{Di};k )</th>
<th>( \text{Ei};r )</th>
<th>( \frac{k}{r} )</th>
<th>Number of initial faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.10;0.80</td>
<td>1;0.90</td>
<td>0.89</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>0.10;0.85</td>
<td>1;0.95</td>
<td>0.89</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>0.10;0.70</td>
<td>1;0.95</td>
<td>0.74</td>
<td>10</td>
</tr>
</tbody>
</table>

Note that for each version, the ratio \( \frac{k}{r} < 1 \). Some authors (Takuno and Yamada, 2003) argue that this ratio can serve as an index of software availability improvement and ratios smaller than one indicate “good software quality” while ratios larger than one indicate “poor software quality”. Comparisons are made both among different single versions and 3-versions. The values of average availability of different systems over time are shown in Table 4.2. Availability plots are shown in Figure 4.3 and Figure 4.4. In industrial practice, the values in Table 4.2 can be gained from experience or from data analysis of previous projects.
### Table 4-2 Average Availability of Different Software Systems in Scenario One

<table>
<thead>
<tr>
<th>Time (t)</th>
<th>Single Version</th>
<th>3-versions ABC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0.96976</td>
<td>0.96974</td>
</tr>
<tr>
<td>2</td>
<td>0.95316</td>
<td>0.95309</td>
</tr>
<tr>
<td>10</td>
<td>0.92478</td>
<td>0.92442</td>
</tr>
<tr>
<td>30</td>
<td>0.92325</td>
<td>0.92280</td>
</tr>
<tr>
<td>50</td>
<td>0.92658</td>
<td>0.92627</td>
</tr>
<tr>
<td>70</td>
<td>0.92993</td>
<td>0.92984</td>
</tr>
<tr>
<td>100</td>
<td>0.93430</td>
<td>0.93443</td>
</tr>
<tr>
<td>120</td>
<td>0.93665</td>
<td>0.93652</td>
</tr>
</tbody>
</table>

**Figure 4-3 Scenario One: Instant Availability**
Chapter IV. Software Availability Assessment of N-version Programming Systems

Figure 4-4 Scenario One: Average Availability

It can be concluded that in this scenario, software availability of the 3-version software system is better than any of its sub-version systems. For each single sub-version program, software availability is affected by failure rate, restoration rate and the number of initial faults. For each single version, the behaviors of versions with similar ratio $\frac{k}{r}$ are also similar, as indicated by sub-version A and B. The availability performance of single versions with smaller ratio $\frac{k}{r}$ is better than that with larger ratios. Availability is greatly increased if the system
changes from single version to N-version and the availability performance of N-version systems are smoother than their single sub-version counterparts with no steep drop or increase in availability overtime. The smooth curve also indicates a stable performance of the software system. In all, the availability performance is greatly increased if N-version programming technique is applied in positive environments.

*Scenario two*

As a comparison to Scenario One, we consider the worst case in this scenario. Suppose three poorly-developed sub-version programs are provided and we would like to check the impact of N-version programming on software availability in this case. Details of each version are provided in the following table.

<table>
<thead>
<tr>
<th>Version</th>
<th>$D_i; k$</th>
<th>$E_i; r$</th>
<th>$\frac{k}{r}$</th>
<th>Number of initial faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.10;0.80</td>
<td>1;0.70</td>
<td>1.14</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>0.10;0.85</td>
<td>1;0.75</td>
<td>1.13</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>0.10;0.70</td>
<td>1;0.60</td>
<td>1.17</td>
<td>10</td>
</tr>
</tbody>
</table>

Note that in this scenario, for each version the ratio $\frac{k}{r} > 1$. This ratio indicates “poor software quality”, as discussed in the last Scenario. Comparisons are made both among different single versions and 3-versions. The values of average availability of different systems over time are shown in Table 4.4. Availability plots are shown in Figure 4.5 and Figure 4.6.
Table 4-0-4 Average Availability of Different Software Systems in Scenario Two

<table>
<thead>
<tr>
<th>Time (t)</th>
<th>Single Version</th>
<th>3-versions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>0.96976</td>
<td>0.96974</td>
</tr>
<tr>
<td>2</td>
<td>0.95316</td>
<td>0.95309</td>
</tr>
<tr>
<td>10</td>
<td>0.91478</td>
<td>0.91442</td>
</tr>
<tr>
<td>30</td>
<td>0.91325</td>
<td>0.91280</td>
</tr>
<tr>
<td>50</td>
<td>0.89658</td>
<td>0.899627</td>
</tr>
<tr>
<td>70</td>
<td>0.89293</td>
<td>0.88984</td>
</tr>
<tr>
<td>100</td>
<td>0.88130</td>
<td>0.87843</td>
</tr>
<tr>
<td>120</td>
<td>0.87165</td>
<td>0.86952</td>
</tr>
</tbody>
</table>

Figure 4-5 Scenario 2: Instant Availability
Chapter IV. Software Availability Assessment of N-version Programming Systems

Figure 4-6 Scenario Two: Average Availability

Similar conclusions as in Scenario One can still be made. Apparent improvement in software availability performance can be observed when the N-version programming technique is involved. The performance of software availability of the 3-version software system is better than any of its single sub-version programs. However, as all the ratios \( \frac{k}{r} > 1 \), the availability is still decreasing with time, but the decrease of software availability in N-version programming system is slower than any of its sub-versions. This is the phenomenon that people try to avoid in real industrial cases.
Scenario Three

In this scenario, a moderate case is considered. Suppose three sub-version programs are provided. One is well-developed while the other two are poor-developed. Details of each version are provided in the following table.

<table>
<thead>
<tr>
<th>Version</th>
<th>$D_i; k$</th>
<th>$E_i; r$</th>
<th>$\frac{k}{r}$</th>
<th>Number of initial faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.10; 0.80</td>
<td>1; 0.90</td>
<td>0.89</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>0.10; 0.85</td>
<td>1; 0.50</td>
<td>1.7</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>0.10; 0.70</td>
<td>1; 0.50</td>
<td>1.4</td>
<td>10</td>
</tr>
</tbody>
</table>

Note that in this scenario, for version A the ratio $\frac{k}{r} < 1$ while for version B and C the ratio $\frac{k}{r} > 1$. As discussed in previous scenarios, A is of “good quality” and B&C are of “poor quality”. We would like to check how N-version programming technique affects the system performance in this case. Comparisons are made both among different single versions and 3-versions. The values of average availability of different systems over time are shown in Table 4.6. Availability plots are shown in Figure 4.7 and Figure 4.8.
Chapter IV. Software Availability Assessment of N-version Programming Systems

Table 4-6 Average availability of different software systems in Scenario three

<table>
<thead>
<tr>
<th>Time (t)</th>
<th>Single Version</th>
<th>3-versions</th>
<th>ABC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>0.96976</td>
<td>0.96969</td>
<td>0.96977</td>
</tr>
<tr>
<td>2</td>
<td>0.95316</td>
<td>0.95275</td>
<td>0.95310</td>
</tr>
<tr>
<td>10</td>
<td>0.92478</td>
<td>0.91187</td>
<td>0.91788</td>
</tr>
<tr>
<td>30</td>
<td>0.92325</td>
<td>0.86492</td>
<td>0.89139</td>
</tr>
<tr>
<td>50</td>
<td>0.92658</td>
<td>0.82241</td>
<td>0.87244</td>
</tr>
<tr>
<td>70</td>
<td>0.92993</td>
<td>0.78205</td>
<td>0.85614</td>
</tr>
<tr>
<td>100</td>
<td>0.93430</td>
<td>0.72633</td>
<td>0.83484</td>
</tr>
<tr>
<td>120</td>
<td>0.93665</td>
<td>0.69425</td>
<td>0.82279</td>
</tr>
</tbody>
</table>

Figure 4-7 Scenario Three: Instant Availability

110
In this scenario, the performance of software availability of the 3-version software system no longer always outperforms its single sub-programs. In fact, the optimal strategy for this scenario is to construct the software system using a single version A. Although the performance of ABC collaboration is better at the beginning, sub-version A will outperform 3-versedioned ABC as time increases. On the other hand, the performance of ABC collaboration is still much better than B and C alone.

Based on the above three different scenarios, it can be concluded that N-version programming technique has very apparent positive impact on software system’s availability.
Chapter IV. Software Availability Assessment of N-version Programming Systems

performance. However, the availability performance of an N-version system is greatly affected by the property of its sub-version programs and larger “N” does not always necessarily guarantee higher availability performance. Concluding from the above three scenarios, our strategy is to choose sub-versions with ratio $\frac{k}{r} < 1$ to construct the N-version system. If the number of sub-versions with ratio $\frac{k}{r} < 1$ is less than half of the required N-versions, the required N-version system with highest availability can never be obtained and a better approach is to reduce the number of required versions. If all of the sub-versions are with ratio $\frac{k}{r} > 1$, such versions are not appropriate for constructing software systems with availability constraints.

For software projects with limited budget and where the availability is not of the main concern, building a single version system with good reliability/availability should be of high priority. For systems with availability constraints, one also needs to look into the properties of different sub-versions. It is not a “the-more-the-better” approach as the interactions among different versions determines the final performance. The availability model proposed in this research provides a possible approach to establish the effects of such interactions and may be used to determine the optimal structure for N-version programming software systems.

4.3.3 Optimal Software Structure under Software Availability and Budget Constraints

In continuation with the work presented in the previous sections, our model is applied to an optimal software structure (Levitin, 2005) problem in this section. For software availability, it is a the-more-the-better metric and for many systems there is a strict requirement of software
availability over the operational cycle. As it is shown in the previous section, N-version programming has very apparent impact on software availability performance, however, we cannot build a software system with infinite versions due to budget constraints and we hope to reduce the total cost during the operational cycle as much as possible. For certain projects, people need to investigate the cost of traditional software and N-version software before deciding which one to apply. This is a typical problem that managers of IT sectors often face. In this section, we present a novel approach for determining the optimal software structure for N-version software systems under software availability and budget constraints.

Optimization modelling

Our optimization is based on the cost-effective basis and a proper cost model is needed to search for optimization. The total cost of an N-version programming software system during its operational cycle usually consists of three main components, namely operation cost, maintenance cost and risk cost.

(1) Operation cost $C_o$. This is the cost incurred for normal software operation. Operation cost usually includes cost of human resource, power supply and usually linearly increases with time. If we denote the expected length of operational cycle as $L$, the operation cost can be expressed as:

$$C_o = c_o L$$

(4.11)
where $c_o$ is the expected operation cost per unit time. This cost is commonly deterministic for most software systems.

(2) Maintenance cost $C_m$. This is the cost incurred for maintaining a software system. Maintenance tasks are usually executed by maintenance teams and different maintenance policies often correspond to the different team sizes. This cost is controllable by adapting different maintenance policies and it can be expressed as:

$$C_m = \sum_{i=1}^{n} c_i L$$

where $c_i$ represents the current maintenance policy and it is the expected maintenance cost per unit time for sub-version $i$. Normally the failure restoration rate $E_i$ of sub-version $i$ is proportional to $c_i$ and the relationship can be denoted as $E_i \propto c_i$. For each maintenance team, there is a minimal team size and we denote the minimal failure restoration rate of the $i$th sub-version, which is corresponded by the smallest maintenance size as well as cheapest maintenance policy, by $E_{i0}$. Without loss of generality, the policy can be written as:

$$\begin{cases}
    c_i = k_i \varphi_i E_{i0} \\
    E_i = k_i E_{i0}
\end{cases} \quad k_i = 1, 2, 3, \ldots$$

(4.13)
where $k_i$ is the decision factor for maintenance policy in version $i$ and $\phi_i$ is a deterministic proportionality factor. Generally speaking, $k_i$ can be interpreted as a multiple of the minimum of the maintenance team size. Then the total maintenance cost can be further expressed as:

$$C_m = \sum_{i=1}^{n} k_i \phi_i E_{i0} L_i; \quad k_i = 1, 2, 3, \ldots \quad (4.14)$$

(3) Risk cost $C_r$. As the name indicates, this is the cost incurred by an unavailable system. For given software systems, the higher the software availability is, the lower the risk cost will be. This cost can be expressed as:

$$C_r = \int_{0}^{L} c_r [1 - A(\tau)] d\tau \quad (4.15)$$

where $c_r$ is the expected cost per unit time if the system is unavailable and $A(\tau)$ is the availability of software system.

(4) Development cost $C_d$: the cost incurred by developing the target software system. This is a fixed cost, and it is expected that the development cost of an N-version programming system is N times higher than the development cost of a single version software system.

The total cost is the summation of the above four components. As we can see, the maintenance cost directly impacts the maintenance policy and it also affects the software availability, thus it indirectly affects the risk cost. More maintenance effort will lead to less risk.
cost, but it also increases the maintenance cost. Hence, there exists an optimal maintenance policy that minimizes the total cost. If the availability requirement is quantitatively marked as $A_{req}$, the optimization problem can be formulated.

**Objective:**

$$\min C_{total} = C_o + C_m + C_r + C_d = C_d + c_0 L + \sum_{i=1}^{n} k_i \phi_i E_i L + \int_{0}^{L} c_r [1 - A(\tau)] d\tau$$

(4.27)

**Subject to:**

$$A(\tau) \geq A_{req}, \forall \tau \in (0, L)$$

$$k_i = 1, 2, 3,...$$

$$n = 1, 2, 3,...$$

The above optimization problem has two decision variables, namely $n$ and $k_i$. $n$ is the major decision variable and it determines the structure of an N-version programming system and it represents the number of sub-versions within the system; $k_i$ represents the maintenance policy under a determined software structure. Once solved, our proposed model can not only find the optimal software structure but also can find the optimal maintenance policy under the optimal software structure.

**A Numerical Example**

To illustrate our proposed solution to the optimal software structure problem, the example in Scenario One is extended. We incorporate some cost factors (Table 4.7) with the
example in Scenario One. We can either choose to build single-versioned software or 3-versioned software with respect to maintenance policies. We aim to select the most appropriate software system with lowest cost over the operation cycle.

Table 4-7 Maintenance Cost Factors

<table>
<thead>
<tr>
<th>Version</th>
<th>$c_o$</th>
<th>$\phi_i$</th>
<th>$E_{i0}$</th>
<th>$C_r$</th>
<th>$L$</th>
<th>$C_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>500</td>
<td>120</td>
<td>100</td>
</tr>
</tbody>
</table>

Depending on the requirement of software availability, we illustrate two pairs of results by numerically solving the proposed model that is given by equation (4.27). The results are shown in Table.8 and the time value of cost is not considered. This group of results is obtained from simulation. The simulation program was run for 10 thousand times and the average value were chosen as the final output.

Table 4-8 Optimal Software Structure and Maintenance Policy

<table>
<thead>
<tr>
<th>$A_{req}$</th>
<th>$n$</th>
<th>$k_i$</th>
<th>$C_{total}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.95</td>
<td>1</td>
<td>$k_c = 2$</td>
<td>3642.4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$k_A = 1; k_B = 1; k_C = 1$</td>
<td>4601.0</td>
</tr>
<tr>
<td>0.97</td>
<td>1</td>
<td>$k_c = 4$</td>
<td>5684.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>$k_A = 1; k_B = 1; k_C = 1$</td>
<td>4601.0</td>
</tr>
</tbody>
</table>

Judging from the above results, we can see that if the availability requirement is 0.95, the optimal software structure is the single-versioned software of version C with maintenance
policy \( k_C = 2 \); if the availability requirement is 0.97, the optimal software structure is the 3-versioned software with maintenance policy \( k_A = 1; k_B = 1; k_C = 1 \).

Although quite demonstrative, meaningful conclusions can still be made based on the above results. For systems with moderate availability requirements, N-version system do not always serve as the best choice from the cost-effective point of view. The availability of traditional software systems can be enhanced to meet the availability requirement by putting more maintenance efforts at a lower cost. Many end user software systems can be classified into this category, such as online inquiry systems. However, if the availability requirement is very high, the N-version systems will surely perform better in both the availability and cost aspects. These conclusions can be generalized for any general N-version programming structure problems.
Chapter 5  Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack

In the previous two chapters, we dealt with software availability problems with the “internal” properties of software systems, such as software failures, software architecture and so on. Our proposed approaches for estimating software availability under different occasions are shown to be effective and efficient. However, the quality of software systems is affected not only by those “internal” factors, external factors such as software maintenance also plays an important role. External factors such as software maintenance are controllable to some extent, but there are also external factors that are totally unpredictable and nearly impossible to control. Malicious software is one of such unpredictable factors. Unlike traditional software systems, malicious software is designed to hamper normal software operation and can greatly affect the effectiveness of the target system. An investigation is conducted in this chapter to explore the impact of malicious software on the quality of software systems. Due to its network structural property, a distributed software system is considered in this chapter as the target software system. The impact of malware attack is first analyzed based on the homogeneous distributed software system and then it is extended to any general distributed software system. We proposed a novel and systematic approach to quantitatively measure the impact of malware attack on software quality in terms of software reliability/availability degradation. Comparisons show clear evidence that malicious software can greatly decrease quality of software systems. Our investigation also provides a further insight on the behavior analysis of malicious software.
5.1 Introduction

As technology and the society develop, the demand for computing has increased. Many super computers have been built to meet such needs, however, these mainframe level computers are expensive and not all people/companies can afford them. In the meantime, many different technologies have been introduced to produce the required computing capacity at affordable costs, and distributed computing is one of such technologies (Kan, 2003).

A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal. A computer software system that runs in a distributed system is called a distributed software system. In distributed computing, a problem is divided into many tasks, each of which is solved by one computer within the distributed system (Andrews, 2000). To summarize, one can conclude that a distributed system is 1) based on certain types of network structure, 2) work is dispatched among different individual and autonomous nodes within the network and 3) different nodes communicate with each other (Ghosh, 2007).

Distributed systems used to refer to computer networks where individual computers were physically distributed within some geographical area (Andrews, 2000). Nowadays more people argue that distributed systems are a much wider concept and even the Internet can be regarded as a certain type of distributed system (Andrews, 2000; Ghosh, 2007). Though different in forms, it has been claimed by many authors that distributed systems can greatly increase the computing capacity at moderate costs (Dai et al, 2008). Besides the increase of computational capacity, the quality issues have also been studied (Dai et al, 2003; Dai and Levitin, 2007). There is apparent software reliability increase in distributed software systems (Dai et al, 2003) on many occasions.
Chapter V. Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack

However, most people just see the positive side of distributed computing and there is one clear factor that people usually ignore when dealing with the quality problems of distributed systems – the vulnerability of its network structure.

Malicious software, or more often called as malware for short, is software designed to infiltrate a computer system without the owner's informed consent. The expression is a general term used by computer professionals to mean a variety of forms of hostile, intrusive, or annoying software or program code (Kondakci, 2008; Microsoft, 2004). According to a leading system and commercial software vendor Microsoft (2004), malware includes computer viruses, worms, Trojan horses, spyware, dishonest adware, crimeware, most rootkits, and other malicious and unwanted software. These types of malware not only harm the target computer system but also may cause serious legal issues.

One major characteristic of malicious software is that it reproduces itself and infects other clean systems whenever possible (Kondakci, 2008, Kondakci, 2009). In the early days of computer industry, malware was spread via floppy disks, which was slow and ineffective (Kondakci, 2008). However, with the development of computer networks, especially the Internet, the spreading speed of malware has exponentially increased and the loss that was associated with such increase was incredible. For example, the Blaster virus that stroked the world in 2003 infected over 1 billion PCs all over the world and the loss that was estimated to be no less than 20 billion US dollars. The threat of malware has never been so real and severe, and the network structure of distributed systems makes it a perfect target for malware attack.

It is natural to assume quality degradation of software system that is infected by malware, however, little has been recorded in this area so far. Existing literature that deals with malware primarily focuses on the infection and removal mechanism (Gutpa et al, 2009; Feng and Gutpa, 2004).
Chapter V. Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack

2009; Kolter and Maloof 2004) while existing literature that deal with software quality degradation (Schultz et al. 2001; Wang et al. 2003) seldom takes malware into consideration. It is necessary to perform an analysis on malware attack over software quality degradation and we are motivated to conduct a thorough study.

To improve our understanding of the malware attack mechanism, we proposed a systematic approach to model the behaviors of malware spread, based on which the quality degradation of the target distributed software system can be measured quantitatively. The rest of this chapter is organized as follows. The structure of homogeneous distributed systems is first analyzed in Section 5.2 and a Markov chain is proposed to model the malware attack behaviors within the system. Malware defense factors are also considered and a basic image of malware attack is obtained. In Section 5.3 we extend homogenous distributed system to general distributed systems and a novel continuum model is proposed to further analyze the malware spreading mechanism. Then software reliability, software availability and software service reliability is explicitly derived in Section 5.4 and a numerical example shows how our proposed method can be applied in practice.

5.2 Distributed Software Systems and Malware Epidemics: The Homogeneous Scenario

The purpose of a distributed system is to coordinate the use of shared resources or provide communication services to the users (Ghosh, 2007). Distributed system often consists of distributed hardware system and distributed software architecture. The distributed software
architecture is of great importance, because without distributed software, the distributed system is of limited use (Reza, 2006).

On the other hand, most failures occurred in distributed system can be classified as software problems. Reliability is an important metric for distributed system performance. Many research efforts have been devoted to the reliability analysis of distributed systems, including both system/software reliability (Huang et al, 2007; Tamaru and Yamada, 2009) and service reliability (Dai et al, 2003) of distributed systems. System availability is another important performance measure for complex systems and there is also plenty of literature on the availability topics of distributed systems (Dai et al, 2004). In summary, most of the related research concludes that distributed system’s structure boosts both its reliability and availability, when compared to traditional computer systems. However, most of the previous studies focused on failures caused by accidents or natural cataclysms. On the other hand, infective virus and malware become widely spread in the current computer networks including many distributed systems. Protecting the distributed system against the malicious attacks becomes an increasingly critical issue.

5.2.1 Modelling of Distributed System under Malware Attack: The Homogeneous Scenario

In general, distributed systems are designed to coordinate node computers that are connected via certain types of network and provide computing services. In a typical distributed system, service request is usually received by a resource management server (RMS) and then the request is divided and dispatched to the computer nodes within the distributed system.
Various hardware and software architectures are used for distributed systems. If identical copies of application software run on the same type of computer nodes within a distributed system, then this kind of system is called a homogeneous distributed system. A homogeneous distributed system is the simplest yet the most widely used distributed system architecture (Andrews, 2000; Ghosh, 2007, Agneeswaran and Janakiram, 2009) with many researches and application studies (Xie et al, 2004). As we would like to have a rough image of malware spreading within distributed software systems, we start our investigation with the simplest scenario in this section. The conceptual structure of a homogeneous N-nodes distributed system is shown in Figure 5.1.

![Figure 5-1Structure of a N-nodes homogeneous distributed system](image)

Since all computer nodes within a distributed system communicate with each other, it is easy for a malware-infected computer node to attack other computers. Malware disturbs normal operation of a computer node, which makes it unable to perform the assigned task anymore. In the mean while, malware utilizes all available resources of the infected computer node to attack other un-infected ones.
Chapter V. Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack

The attack of malware takes time and consumes resources such as CPU time and network bandwidth. Such malicious behavior deteriorates the overall performance of the distributed system. However, not all malware attacks are successful. The probability of a single successful malware attack is not high, but if the large amount of all target computer nodes is taken into consideration, some “unlucky” computer nodes still will get infected. In other words, if improper actions are taken, all nodes will ultimately become infected even if only one computer node was infected at the very beginning.

Luckily, complex systems such as distributed systems are usually taken good care of by professionals. Routine checking mechanisms are often deployed so that malfunctioning components can be restored to normal state. This can be done either by manually or by security software systems such as antivirus software. In order to model the behavior of malware attacks and these restoring performances, a continuous Markov chain is built with proper assumptions.

We made the following assumptions to facilitate our analysis in this research:
Assumptions:

1. A homogeneous N-nodes distributed system consists of N identical computer nodes, on which identical software applications are installed. Malware keeps attacking the system from the outside.
2. Each computer node has the same service rate $\alpha$. The work of each individual node is independent.
3. Each system node has an initial failure rate of $\lambda_i$ and an initial infection probability $\lambda_m$ that one computer node will fail or get infected, respectively. If a computer node is infected by malware, it can no longer perform the assigned task anymore. In the mean while, it keeps attacking other un-infected nodes with infection rate $\lambda_a$.
4. Routine system checking is performed. After checking, a node can be restored with probability $\mu_r$. If both infected nodes and failed nodes exist, failed nodes are dealt with first.
5. Only one un-infected node can be infected by malware attacking at a time.

All the above assumptions are easy to validate except for Assumption 2. There are cases where certain nodes have to wait for the output of others as their input. Dai et al. (2006; 2008) discussed the problems of correlated nodes in distributed/grid systems. Since it is hard to estimate the level of correlation, which is normally very low (Dai et al., 2004) and our main focus is to check the impact on quality degradation caused by malware attack, this assumption is neglected in our study.
Consider a stochastic process \( \{X(t)\} \), where \( X(t) = \{H, I, D\} \) is the state of an \( N \)-node distributed system. \( H \) denotes the number of healthy nodes, \( I \) denotes the number of infected nodes and \( D \) denotes the number of failed nodes. It can easily be proven that the Markov property holds in this process and \( \{X(t)\} \) can be modeled as a continuous time Markov process. Due to the size limit of this page, we only present the transition graph of a single state \( \{H, I, D\} \).

Suppose that at least one computer node is infected by malware, and then the state transition diagraph is obtained in Figure 5.2.

![Figure 5-2 Transition graph of state \( \{H,I,D\} \)](image_url)

Based on the transition graph in Figure 5.2, when there are infected nodes, for the state \( \{H, I, D\} \) we could obtain a Kolmogorov differential equation listed below and \( p_{\{H,I,D\}}(t) \) stands for the probability at time \( t \) that the system is in state \( \{H, I, D\} \):

\[
\begin{align*}
\mu r & \quad (H+1, I-1, D) \\
\lambda i & \quad (H, I-1, D+1) \\
\mu r & \quad (H-1, I+1, D) \\
H\lambda i & \quad (H-1, I, D+1) \\
H\lambda i & \quad (H, I, D) \\
(H+1)\lambda m+(I-1)\lambda a & \quad (H+1, I-1, D)
\end{align*}
\]
The differential equations can be obtained similarly on other states. By consideration all the states, we can have a set of differential equations describing the transitions between different states.

If no malware infection has ever been reported, another type of equation can be obtained for the special state \( \{H,0,D\} \):

\[
(H_{\lambda_r} + H_{\lambda_m}) \frac{d}{dt} p_{(H,0,D)}(t) = \mu_r p_{(H-1,0,D+1)}(t) + \mu, p_{(H-1,1,D)}(t)
\]

Similar to equation (5.1), we can obtain a set of differential equations to describe the transitions between states without considering the malware infections.

To solve the equation systems based on equation (5.1) and equation (5.2), we have to obtain the boundary conditions. We assume that the system starts from an all-healthy state, hence the boundary conditions are:

\[
\sum p_{(H,I,D)}(t) = 1, p_{(N,0,0)}(0) = 1, p_{(N-1,0,1)}(0) = 0, \ldots, p_{(0,N,0)}(0) = 0, p_{(0,0,N)}(0) = 0
\]
The continuous Markov process can be explicitly solved with equations (5.1) ~ (5.3) and then we are able to get the transient state probability \( p_{(H,I,D)}(t) \) of each state, which will greatly improve our capability of analyzing the system.

### 5.2.2 Derivation of Software Service Reliability and Software Availability

The analysis in the previous section is meant for obtaining meaningful software quality metrics such as software reliability, software service reliability and software availability. Since software reliability of distributed systems has been studied by many authors and there is plenty of literature to refer to (Xie et al, 2004), it will not be covered in this section.

Only a few researchers have ever considered the service reliability of distributed systems. There is still no precise definition of service reliability, but most authors (Dai et al, 2006) considered it as the probability of successfully completing a target task within a required time interval. In other words, the service reliability is the probability that a task will be finished in time and with correctness. Apparently, the underlying criterion of software service reliability is software reliability. In this research, we also adopt this definition. According to our definition of service reliability, the service reliability \( R_s(t) \) of our target system at time \( t \) is the probability that it can complete the requested service within time \( T_E \):

\[
R_s(t) = \Pr\{T \leq T_E\} \quad (5.4)
\]
We further assume that the completion time of a certain requested service follows an exponential distribution. Then equation (5.4) can be rewritten as

\[ R_s(t) = \sum R_s(t \mid \{H, I, D\}) \Pr\{H, I, D\} = \sum p_{(H, I, D)}(t) F(\alpha T_E) \quad (5.5) \]

where \( F \) is the CDF of an exponential distribution and \( \alpha \) is the service rate of a single node. The service availability can be defined as the probability that at least one computer nodes in the distributed system is still functioning:

\[ A_s(t) = \Pr\{H > 0 \mid t\} = \sum_{\forall H > 0} p_{(H, I, D)}(t) \quad (5.6) \]

The above two equations give explicit forms of the transient service reliability and availability at a certain time. However, in most cases people are more interested in the situation when the system transition becomes stable. The steady state service reliability and availability can also be obtained with our proposed model.

**Lemma 5.1:** The steady state probability of the Markov chain proposed in Section 5.2.1 exists.

**Proof:** Since all states communicates with each other, the Markov chain is irreducible and positive recurrent. Thus Lemma 5.1 is proven \( \Delta \)

With Lemma 5.1, the steady state probability can be obtained by the following equations:
where \( p_{(H,I,D)} \) is the steady state probability of state \( \{H,I,D\} \). Then the steady state service availability can be derived as:

\[
A_s = 1 - \sum_{\forall I,D} p_{(0,I,D)}
\]  

(5.8)

And the steady state service reliability can be derived as:

\[
R_s = \sum_{\forall H,I,D} p_{(H,I,D)} F(H\alpha T_E)
\]  

(5.9)

5.2.3 Numerical Examples

In the previous section, a Markov chain model is proposed to model the malware epidemic within a homogeneous distributed software system. In this section, numerical examples will be presented to show how the proposed model can be applied in practice.

Kondakci (2008) conducted a similar study on malware analysis and provided some real data. However, his approach is different from ours and his data is based on single personal computers in an internet environment and may not be suitable for our study. For the purpose of illustration, some artificial data are generated and a 3-node homogeneous distributed system is considered. We first conduct an experimental numerical analysis of service reliability and
availability of the system without considering malware attack. Then we take the malware attack scenario into consideration. The results are compared and meaningful conclusions are made. Although we do not use real case data, our model and analysis approach can be adopted when real datasets are available. Summary of the data is provided in Table 5.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_i$</td>
<td>0.05</td>
</tr>
<tr>
<td>$\mu_r$</td>
<td>0.8</td>
</tr>
<tr>
<td>$\lambda_m$</td>
<td>0.1</td>
</tr>
<tr>
<td>$\lambda_a$</td>
<td>0.3</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.5</td>
</tr>
<tr>
<td>$T_E$</td>
<td>2</td>
</tr>
</tbody>
</table>

If the distributed system is well protected and free of malware attack, then a duplet $\{H, D\}$ is enough for describing the system state. We define state 0~3 as $\{3,0\}$, $\{2,1\}$, $\{1,2\}$ and $\{0,3\}$, respectively. If the distributed system is under malware attack, we have a total of 9 states as $\{3,0,0\}$, $\{2,1,0\}$, $\{2,0,1\}$, $\{1,2,0\}$, $\{1,1,1\}$, $\{0,2,1\}$, $\{0,1,2\}$, $\{0,0,3\}$ and $\{0,3,0\}$, respectively. By solving equations (5.1) ~ (5.3), we could obtain their transient state probability, based on which the system service reliability and availability can be obtained. Then the service reliability and availability can be plotted as follows:
By comparing the different sets of results that we obtained in analyzing two different scenarios, some meaningful conclusions can be made. An apparent decline of both service reliability and availability is observed if malware attack exists. This result indicates that the capability of a distributed system is obviously hampered by malware attack. Further comparison shows that the service reliability has been degraded more than availability has. In fact, even a distributed system is attacked by malware; it is hard for all its computer nodes to get infected at the same time. However, since the computational capability of a distributed system depends
highly on the available computer nodes, it is expected that the service time for a certain task increases rapidly if malware attack exists.

5.3 Distributed Software Systems and Malware Epidemics – A Revisit

In the previous section, we proposed a Markovian model of the malware epidemics within homogeneous distributed software systems. Some preliminary results are obtained and indicate the negative impact of malware on software quality. However, there is still much to improve.

Firstly, the assumption of homogeneous structure is too strong. Although many existing works (Xie et al, 2004; Levitin, 2004) that focused on distributed systems are based on the assumptions of homogeneous structure, it is merely a simplification of the real world scenario. In fact, computer nodes within a distributed system can hardly be classified as “homogeneous” since the processing capability depends highly on the status of each individual computer. What is more, distributed systems in commercial use are usually supported by several powerful workstations that connect many other less powerful computers. Algorithms are designed in the way that tasks that require much computational efforts are assigned to these powerful workstations while tasks that require less computation efforts are dispatched to other less powerful computers (Reza, 2006) so that maximum efficiency can be obtained. As such, the assumption of homogeneous structure must be relaxed if further investigation of distributed systems is required.

Secondly, Markovian approach is easy to understand but it is effort consuming. The Markovian approach is suitable when the target system is not very complicated. When the
number of nodes within the distributed system increases, the computational efforts required to solve the Markov chain grows dramatically.

To overcome the above stated shortcomings, we need to reconsider the structure of distributed software systems and the modeling of malware epidemics. In this section, the malware epidemics of distributed software systems are expanded in terms of generalization and efficiency. To better represent a distributed system, in this section the distributed system is modeled as an undirected graph $G = (V, L)$, where $V = \{v_i \mid 1 \leq i \leq N\}$ is the set of independent computers (nodes) within the distributed system, and $L = \{l_{ij} \mid 1 \leq i \leq N, i \leq j \leq N\}$ is the set of communication channels (arcs) connecting the computers.

### 5.3.1 A Continuum State Reliability Model of Individual Nodes

Traditional approaches for modeling a system process with a number of nodes are to use the Markov chain (Xie et al, 2004). The nodes are often modeled to be either online (functioning state) or offline (malfunctioning state) and a stochastic process is brought in to model the online nodes or offline nodes. The Markovian approach is easy to understand, however, the mathematical complexity would increase fast if the target system is large and complicated (Pham, 2003). On the other hand, when a distributed system under virus infection is concerned, it is impractical to assert that nodes can only be either online or offline. In fact, a virus-infected node will not immediately lose its computational capability. A virus-infected node will continue trying to infect other uninfected nodes and will either gradually lose its computational capability or will recover to healthy state with the help of virus defense mechanisms such as anti-virus
programs and system patches. As such, a two-state model is not suitable for this research and a more appropriate index is needed to represent the states of individual nodes.

Possible solutions include the multi-state model with discrete states and continuum model. As pointed out by Brunelle and Kapur (1999), the key issue for adapting a continuum model is to determine whether it is feasible or necessary to use a continuum model because the discrete multi-state model is simpler and it can also handle a wide range of problems. In the case of virus spreading, the state of one processor/node is strongly influenced by the states of its neighbors. To model the virus spreading dynamics, differential equations are used to solve dynamics of virus spreading (Murray 1988, Buzna et al. 2006, Wierman and Marchette 2004) and the differential equations require the state variable to be differentiable (or continuous in this case). In the reliability literature, continuous state models have been studied by a number of researchers. Baxter (1984) first introduced the standard continuum reliability model that allows the state of a system or a component to take any real value in the closed interval [0, 1]. The continuous reliability model has been an ongoing research area over the past decades. Some recent publications relevant to continuous state model are Liu et al. (2003), Long et al. (2008), Yi and Lisnianski (2008). In fact most real-world systems and components exhibit the continuous degradation in some form. For example, an automobile brake gradually loses its braking power as the car keeps traveling. Similarly, a virus-infected computer gradually loses its computation capability as the virus keeps consuming resources and the continuous degradation is perfect to describe such lose.

The continuous-state model has a range of values \( \Omega = [0, 1] \) representing all possible intermediate performance levels between the two extremes: ‘0’ which is perfect functioning state
and ‘1’ which is the complete failure state. The state value of a node (or computer) \( v_i \): \( x_i(t) \) is regarded as a continuous stochastic process that takes the value from the state space \( \Omega \) and is related to the mission time \( t \). The probability density function of \( x_i \): \( f(x_i,t) \) governs the stochastic behavior of the random state. By the definition of probability, we have \( \int_0^1 f(x_i,t) dx_i = 1 \). Although the process is random, the deterministic trend of the process can be captured by the expectation of the state \( \mu_i(t) = E[x_i(t)] \). By following the study of Brunelle and Kapur (1999), we assume that at the mission time \( t \), the state follows a normal distribution (or Gaussian process). Let \( C_N \) denote a cumulative distribution function for a normal distribution.

\[
C_N(s, \mu(t), \sigma(t)) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} \exp \left( -\frac{(x(t) - \mu(t))^2}{2\sigma^2} \right) dx
\]  

(5.7)

and \( C_{NT} \) represents the same function truncated into the interval \([0, 1]\):

\[
C_{NT}(s, \mu(t), \sigma(t)) = C_{NT}(s, \mu(t), \sigma(t)) - C_{NT}(0, \mu(t), \sigma(t)) \]

\[
= \frac{C_{NT}(1, \mu(t), \sigma(t)) - C_{NT}(0, \mu(t), \sigma(t))}{C_{NT}(1, \mu(t), \sigma(t)) - C_{NT}(0, \mu(t), \sigma(t))}
\]

(5.8)

where \( s \) is the state index, \( \mu_i(t) \) is the expected state of node \( i \) and \( \sigma_i(t) \) is the standard deviation of the state of node \( i \). It is noted that they are all functions of the mission time \( t \). In the next section, we apply the epidemic functions to solve \( \mu_i(t) \). Moreover, we assume that \( \sigma_i(t) \) is related to \( \mu_i(t) \), in particular, we assume that \( \sigma_i(t) \) will increase to its maximum when
\( \mu_i(t) = 0.5 \) and then gradually decrease (Brunelle and Kapur, 1999). Therefore, we have the following definition of \( \sigma_i(t) \) for this thesis.

\[
\sigma_i(t) = 0.25(0.5 - |0.5 - \mu_i(t)|) + 0.001 \tag{5.9}
\]

### 5.3.2 Virus Epidemic Model: The General Scenario

Judging from the assumptions made in Section 5.3.1, if a node \( v_i \) is in a healthy state, then \( \mu_i(t) = 0 \). The deviation from the normal state represents the level of damage to the system. The strength of interactions between node \( v_i \) and its neighbors \( v_j \) is defined by weight \( w_{ij} \) attributed to the edge \( e_{ij} \). In this thesis the weight is represented by the speed of the connection channel between two nodes. An infected node \( v_i \) has an impact on all neighbors \( v_j \) proportionally to the connection strength \( w_{ij} \) with some time delay \( t_{ij} \). For node \( v_i \), its status also depends on its own ability to defend virus infection. Taking all the above factors into consideration, the status change of node \( v_i \) can be represented using the following epidemic differential equation:

\[
\frac{d}{dt} \mu_i(t) = w_{ij} \sum_{j \in \Phi_i} \mu_j(t - t_{ij}) - \mu_i(t) \delta_i \tag{5.10}
\]

where \( \Phi_i \) is the set of nodes that connect node \( i \) and \( \delta_i \) is the time independent parameter that represents the ability of individual nodes to defend virus infection. \( \delta_i \) is an important parameter
with concrete physical meaning. For example, the anti-virus software such as McAfee usually owns a series of products offering different levels of protections ranging from basic to total protection. The more protections the anti-virus offers; the higher priced it is. This parameter is determined by assuming the fullest protection service to be 1 and the other smaller protection schemes are the fractions of it.

Based on the single epidemic equation in (5.10), we establish the following equation system to model the dynamics of virus spreading in the entire distributed network.

\[
\begin{align*}
\frac{d}{dt} \mu_1(t) &= w_{1j} \sum_{j \in \Phi_i} \mu_1(t-t_{1j}) - \mu_1(t) \delta_1 \\
&\vdots \\
\frac{d}{dt} \mu_i(t) &= w_{ij} \sum_{j \in \Phi_i} \mu_j(t-t_{ij}) - \mu_i(t) \delta_i \\
&\vdots \\
\frac{d}{dt} \mu_N(t) &= w_{Nj} \sum_{j \in \Phi_N} \mu_N(t-t_{Nj}) - \mu_N(t) \delta_N
\end{align*}
\]

There are several routines to solve differential equation systems such as equation (5.11). The widely accepted Laplace transformation approach \( L\{\mu(t)\} = \int_0^\infty \mu(t)e^{-st}dt \) is an effective approach that has a very attractive property that \( L\{\mu'(t)\} = sL\{\mu(t)\} - \mu(0) \). By taking the Laplace transform of both sides of each equation in (5.11), the differential equation system can be numerically solved provided that some initial conditions \( \mu_i(0) = 1, \mu_j(0) = 0, j \neq i \) are given. If the expectations of nodes’ status are solved, the stochastic model of virus spreading (in equation (5.7)) can be easily obtained. An illustrative example will be presented in the next section.
5.3.3 An Illustrative Example

In this section, an example of how to estimate the status of a distributed system under virus attack is presented. It is assumed that a distributed system with 5 nodes has been infected by a software virus. Edge weight and virus defense parameter values are provided in Table 5.2. Network topology of the distributed system is provided in Figure 5.4. The resource management server (RMS) is not presented in the network topology since it connects every node For the purpose of illustration, in this numerical example the edge weight and node virus defense parameters are assigned with some preset values. In industrial practice, such parameters can be easily obtained and replaced by field data. The edge weight is estimated by the level of data interaction between different nodes, which can be monitored by data flow while the virus defense parameters can be estimated by the anti-virus software installed in each node, firewall security level configuration and so on.

<table>
<thead>
<tr>
<th>Edge</th>
<th>$e_{12}$</th>
<th>$e_{13}$</th>
<th>$e_{15}$</th>
<th>$e_{23}$</th>
<th>$e_{34}$</th>
<th>$e_{55}$</th>
<th>$e_{45}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>0.04</td>
<td>0.09</td>
<td>0.03</td>
<td>0.01</td>
<td>0.06</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td>Node</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defense</td>
<td>0.2</td>
<td>0.05</td>
<td>0.2</td>
<td>0.125</td>
<td>0.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 5-4 illustrates a simple example of virus spreading in a distributed system with 5 nodes. Suppose that node 1 is completely infected at system time $t = 0$ (node 1 is the source of virus), we have the initial conditions $\mu_1(0) = 1, \mu_2(0) = 0, \mu_3(0) = 0, \mu_4(0) = 0, \mu_5(0) = 0$ and the following set of differential equations. Because the size of a virus (about several Kbits) is often very small compared to the bandwidth of the connection (which is usually measured in Mbits or Gbits), the time delay caused by network traffic is often negligible. Based on the conditions given above, an initial form of virus epidemic equations can be obtained.
\[
\begin{align*}
\frac{d}{dt} \mu_1(t) &= 0.04 \mu_2(t) + 0.09 \mu_3(t) + 0.03 \mu_4(t) - 0.2 \mu_1(t) \\
\frac{d}{dt} \mu_2(t) &= 0.04 \mu_1(t) + 0.01 \mu_3(t) - 0.05 \mu_2(t) \\
\frac{d}{dt} \mu_3(t) &= 0.09 \mu_1(t) + 0.01 \mu_2(t) + 0.06 \mu_4(t) + 0.02 \mu_5(t) - 0.2 \mu_3(t) \\
\frac{d}{dt} \mu_4(t) &= 0.06 \mu_3(t) + 0.05 \mu_5(t) - 0.125 \mu_4(t) \\
\frac{d}{dt} \mu_5(t) &= 0.03 \mu_1(t) + 0.02 \mu_3(t) + 0.05 \mu_4(t) - 0.1 \mu_5(t)
\end{align*}
\]

Let \( M_i(s) \) denote \( L\{\mu_i(t)\} \), the Laplace transform of (6) can be written as:

\[
\begin{align*}
sM_1(s) &= 0.04 M_2(s) + 0.09 M_3(s) + 0.03 M_4(s) - 0.2 M_1(s) \\
sM_2(s) + 1 &= 0.04 M_1(s) + 0.01 M_3(s) - 0.05 M_2(s) \\
sM_3(s) + 1 &= 0.09 M_1(s) + 0.01 M_2(s) + 0.06 M_4(s) + 0.02 M_5(s) - 0.2 M_3(s) \\
sM_4(s) + 1 &= 0.06 M_3(s) + 0.05 M_5(s) - 0.125 M_4(s) \\
sM_5(s) + 1 &= 0.03 M_1(s) + 0.02 M_3(s) + 0.05 M_4(s) - 0.1 M_5(s)
\end{align*}
\]

This linear system of equations can be solved in inverse Laplace transform applied to the solution, the solutions to equations (5.13) are presented as follows:

\[
\begin{align*}
\mu_1(t) &= 0.43e^{-0.304t} + 0.32e^{-0.162t} + 0.10e^{-0.146t} + 0.01e^{-0.050t} + 0.15e^{-0.013t} \\
\mu_2(t) &= -0.05e^{-0.304t} - 0.13e^{-0.162t} - 0.06e^{-0.146t} + 0.03e^{-0.050t} + 0.20e^{-0.013t} \\
\mu_3(t) &= -0.46e^{-0.304t} + 0.15e^{-0.162t} + 0.16e^{-0.146t} - 0.01e^{-0.050t} + 0.15e^{-0.013t} \\
\mu_4(t) &= 0.17e^{-0.304t} - 0.40e^{-0.162t} + 0.08e^{-0.146t} - 0.02e^{-0.050t} + 0.16e^{-0.013t} \\
\mu_5(t) &= -0.06e^{-0.304t} + 0.12e^{-0.162t} - 0.23e^{-0.146t} - 0.02e^{-0.050t} + 0.18e^{-0.013t}
\end{align*}
\]
Since $\mu_{i}(t) = E(x_{i}(t))$ and $x_{i}(t)$ follow a normal distribution whose mean and standard deviation is already known, properties of $x_{i}(t)$ can be derived. As can be derived from equations (5.14) $\lim_{t \to \infty} \mu_{i}(t) = 0$, which implies that the distributed system would end up in a totally healthy state in the long run. Such a conclusion can be attributed to the continuous efforts of virus defense. However, in most occasions people are more eager to know the transient state of the system which provides more meaningful information for decision making. As such, the focus of this study is on a certain time period of the target system. The plot of $\mu_{i}(t)$ over time shows the trend of the expected behaviors of each node.
As can be seen from Figure 5.5, the mean value of the state index of node 1 first drops steeply from 1 with time and then it begins to reduce more smoothly. This phenomenon can be attributed to the existence of the virus defense mechanism. The curves of nodes 2, 3, 4 and 5 are similar. They all start from 0 and gradually increase. The node which has larger weighted edges connected to node 1 is expected to be infected faster than the others in the beginning and this
situation is also validated in Figure 5.5 in forms of a steeper increasing curve. It is also worth noticing that the mean index value of each node becomes relatively stable as time increases.

Since the expected status of each single node can be monitored in the above proposed model, we can also derive other meaningful metrics for estimating the performance of the distributed system. In the next section, a model for estimating software service reliability will be presented.

### 5.4 Service Reliability Modeling of Distributed System Considering Malware Epidemics

Based on the deterministic epidemic model of virus spread in Section 5.3, this section derives service reliability of the distributed computing system under the context of virus spreading. Reliability of distributed systems has been studied by many authors, but only a few have considered the service reliability of distributed systems. There is still no precise definition of service reliability, but most active authors (Dai et al. 2003, Dai and Levitin 2008) considered it as the probability of successfully completing a target task within a required period of time. In this research, we also adopt this definition into the context of virus epidemic. Prior to the software service reliability model, the assumptions for the task processing and transmission are presented as follows:

**Assumptions**

1. In a distributed system, a task is first divided into several sub-tasks and processed by a number of resources simultaneously.
2. Each node starts to execute the assigned task immediately after it gets all the necessary inputs. Each node has a data processing speed. This speed is related to the state of the node at the time when all data enters the targeting node.

3. Each link has a data transmission speed (bandwidth) and a failure rate. Each link is statistically independent from all other links and all the nodes in the system. The data transmission time at each node is negligible.

4. The data is transmitted through a same set of links before and after the execution of the sub-task at one node.

5.4.1 Service Reliability Model without Communication Channel Failure

This section derives the service reliability without considering communication channel failures. Suppose that the entire task is divided into \( K \leq N \) sub-tasks (\( N \) is the total number of nodes in the network), and each sub-task is executed in one distinct node in the network. Let \( D \) denote the entire amount of data that has to be forwarded to processing nodes and back to the initiating node, and \( d_k \) denote the amount of data related to sub-task \( k \), we have

\[
D = \sum_{k=1}^{M} d_k
\]

(5.15)

It is noted that \( d_k \) contains two different sets of data: 1) the raw data forwarded to the processing node for processing; 2) the output data of the processing node which has to be sent back to the initiating node. Let \( \xi_k \) denote the percentage of the raw data in sub-task \( k \), then
is the amount of raw data to be processed and \((1 - \xi_k) \cdot d_k\) is the amount of result data to be returned to the initiating node,

The processing speed of node \(v_i\): \(\theta_i\) is negatively related to the state of node \(i\) at time \(t_i\). We have the following relationship:

\[ \theta_i = \alpha_i (1 - x_i(t_i)) \quad (5.16) \]

where \(\alpha_i\) is a constant that links the processing speed to the resource state, and \(t_i\) is the system time when a task enters node \(v_i\) (In a later part of this section we will show that \(t_i\) is actually equal to the forwarding transmission time). Based on the processing speed, we can obtain the execution time of sub-task \(k\) at node \(v_i\) as follows:

\[ T_{ki}^e = \frac{\beta_k \cdot \xi_k \cdot d_k}{\theta_i} \quad (5.17) \]

where \(\beta_k\) is a constant that relates the size of a subtask to the computational complexity of a subtask. The service time includes both task execution time and the transmission time. The data is transmitted between node \(v_i\) and the node that initiates the subtask. According to this assumption, the data is transmitted through the same set of links \(L_k\) before and after the execution of the sub-task at one node. Then the data transmission speed for sub-task \(k\) transmitted to resource \(v_i\) is
where $l_j$ is the bandwidth of the $j$th link in the link set $L_k$. Hence the transmission time of the sub-task $k$ is obtained as follows:

$$T_{ki} = \frac{d_k}{s_k}$$ (5.19) 

Equation (5.19) also implies that it will take $\xi_k T_{ki}'$ units of time for the data of sub-task $k$ to be transmitted to node $v_i$ for processing. Therefore the total time needed to complete the sub-task $k$ at node $v_i$ is $\tau_{ki} = T_{ki} + T_{ki}'$. Let $\varepsilon_k$ denote the node set that processes sub-task $k$. If any node in $\varepsilon_k$ finishes sub-task $k$ then sub-task $k$ is regarded to be completed in $\varepsilon_k$. The entire task is completed once all of its sub-tasks are completed. Denote $T_{\varepsilon_k}$ as the finishing time of sub-task $k$, therefore, the total time needed to process the entire task has the following form:

$$\Psi = \max_{1 \leq k \leq K} T_{\varepsilon_k} = \max_{1 \leq k \leq K} \left( \min_{1 \leq i \leq |\varepsilon_k|} \left( \frac{\beta_k \xi_k d_k}{\alpha_i (1 - x_i (\xi_k T_{ki}'))} + \frac{d_k}{s_k} \right) \right)$$ (5.20) 

where $|\varepsilon_k|$ stands for the number of nodes in the set $\varepsilon_k$. According to our definition of service reliability, the service reliability of the target system at time $t$ is the probability that it can finish the requested service within $T_{E}$ time:
Chapter V. Quality Degradation Analysis of Distributed Software Systems Considering Malware Attack

\[ R(t) = \Pr\{ \Psi \leq T_E \} \quad (5.21) \]

It is noted that the service reliability in equation (5.21) is obtained by assuming that all communication channels are in perfectly reliable condition, which is the ideal case. In the next section, we will include the communication channel failures into the computation of service reliability.

### 5.4.2 Service Reliability Model with Communication Channel Failure

According to the assumptions, the total number of nodes \( N \) in the network is larger than the total number of sub-tasks \( M \) because in many practical cases some nodes have the same sub-tasks to execute and this type of redundancy is necessary to increase the reliability of distributed computing. Let \( \mathcal{E}_k \) denotes the set of nodes that process task \( k \). Then we have the following relationship.

\[
\sum_{k=1}^{N} |\mathcal{E}_k| = N \quad (5.22)
\]

where \( |\mathcal{E}_k| \) is the number of elements in set \( \mathcal{E}_k \). Generally, the set of links and nodes used to complete the entire task forms a task spanning tree (Puder et al. 2006). The task spanning tree can be regarded as a combination of the minimal task spanning trees (MTST) (Dai et al. 2007).
which is a minimal possible set of elements (nodes and links) that guarantees the success of task execution. Any failure of the element will result in the failure of the entire task execution. Therefore, the maximum total number of minimal task spanning tree (MTST) is

$$\max(H) = \prod_{k=1}^{K} |E_k|$$  \hspace{1cm} (5.23)

The actual value of $H$ depends on the topology of the target network and the maximum holds if the topology of the network is a complete graph. A task spanning tree completes the entire task if all of its elements do not fail by the maximal time needed to complete all the sub-tasks.

To compute the service reliability of the entire system, we need to find out all the task spanning trees and obtain the probabilities that the spanning trees fail. Classical algorithms such as depth-first search and breath-first search can be applied to find all spanning trees in an arbitrary network. For one task spanning tree, it fails to complete the entire task if one of its edges fails. According to our assumption, the probability $p_{ij}$ of any link $l_{ij}$ functioning in the network is known and independent from other links. Let $\pi_h$ denote the set of links in a spanning tree $h$, the probability that tree $h$ fails is defined as follows:

$$P_h = 1 - \prod_{l_{ij} \in \pi_h} p_{ij}$$  \hspace{1cm} (5.24)
Let $E_h$ denote the event that the $h$-th spanning tree is functioning, then the probability that at least one spanning tree is working can be obtained by the union of the events $P(E_1 \cup \ldots \cup E_H)$. This probability can be obtained by the disjoint approach presented below:

\[
P(E_1 \cup \ldots \cup E_H) = P(E_1 \cup E_1^cE_2 \cup E_1^cE_2^cE_3 \cup \ldots \cup E_1^c \ldots E_{H-1}^cE_H)
= P(E_1) + P(E_1^cE_2) + P(E_1^cE_2E_3) + \ldots + P(E_1^c \ldots E_{H-1}^cE_H)
\]

(5.25)

where the first item in the right hand side is the probability that the first spanning tree is functioning correctly, the second item is the joint probability that the first MTST is failed but the second spanning tree is working, and so on. In the next section, we will provide our algorithm to compute this probability.

By incorporating the service reliability without linkage failures in (5.21), the service reliability for the entire system is:

\[
R_s(t) = P(E_1)P(\Psi_1 \leq T_E) + \ldots + P(E_1^c \ldots E_{H-1}^cE_H)P(\Psi_H \leq T_E)
\]

(5.26)

### 5.4.3 Computation of Service Reliability

As discussed in the section above, it is difficult to drive a closed form solution for the service reliability therefore in this section we propose an algorithm to compute the service reliability. Section 5.4.1 and Section 5.4.2 show that the calculation of total system reliability includes two parts: the equations (5.21) and (5.25).
The entire service consists of $H$ minimal task spanning trees (MTST), for one MTST $\Lambda_h$, we have the following formula to compute its task completion time.

$$\Psi_h = \max_{1 \leq k \leq K, j \in \Lambda_h} \left( \frac{\beta_k \xi_k d_k}{\alpha_i (1 - x_i (\xi_k T_{ki}^j))} + \frac{d_k}{s_k} \right)$$

(5.27)

By definition, the probability that MTST $\Lambda_h$ can complete the entire task within the expected time $T_E$ is:

$$P(\Psi_h \leq T_E) = P\left( \max_{1 \leq k \leq K, j \in \Lambda_h} \left( \frac{\beta_k \xi_k d_k}{\alpha_i (1 - x_i (\xi_k T_{ki}^j))} + \frac{d_k}{s_k} \right) \leq T_E \right)$$

$$= \prod_{k=1}^{K} P\left( \frac{\beta_k \xi_k d_k}{\alpha_i (1 - x_i (\xi_k T_{ki}^j))} + \frac{d_k}{s_k} \leq T_E \right)$$

(5.28)

It is noted that in equation (5.28) $x_i (\xi_k T_{ki}^j)$ is a random variable that follows the truncated normal distribution in equation (5.8). After several algebraic manipulations, can be rewritten as:

$$P(\Psi_h \leq T_E) = \prod_{k=1}^{K} P\left[ x_i (\xi_k T_{ki}^j) \leq 1 - \frac{\beta_k \xi_k d_k}{\alpha_i (T_E - d_k / s_k)} \right]$$

(5.29)

The above probability can be easily obtained by using equation (5.8). The next step is to compute the probability that at least one MTST is functioning. To compute this probability,
disjoint products approach is used (as shown in equation (5.25)). First, all MTSTs are sorted in ascending order of the number of links (as all MTSTs have equal number of processing nodes). Let $\Lambda_s^*$ denote the sorted set of MTSTs, $l_i^{h*}$ denote the $i$th link in the $h*$th MTST in the set $\Lambda_s^*$, $e_i^{h*}$ denote the event that the link $l_i^{h*}$ is functioning ($\bar{e}_i^{h*}$ denotes the event that link $l_i^{h*}$ is failed), $p_i^{h*}$ denote the probability that link $l_i^{h*}$ is functioning, and $E_{h*}$ denote the event that the entire $h*$th MTST is functioning. It is worth noting that the event $E_{h*}$ can be represented by a Boolean product of its element event: $E_{h*} = [e_1^{h*} \cap \cdots \cap e_{n_{h*}}^{h*}]$ (where $n_{h*}$ is the total number of links in the $h*$th MTST).

Based on the definitions introduced above, we present an algorithm that can be easily deployed on computers to calculate the service reliability of the entire system (Please refer to Appendix for details.).

### 5.4.4 A Numerical Example

To illustrate the impact of software virus spreading within a distributed computer system on its service reliability, the example in Section 5.3.3 is extended. The service reliability of an ideal system that is free of virus will be derived first, followed by a comparison with the service reliability of a malware-infected system.

Suppose each communication channel between different nodes now has certain probability to fail and the resource management server (RSM) will divide input task into two sub-tasks and then dispatch them. In real industrial practice, the algorithm for determine task dispatch is complicated. Andrews (2000) listed several popular task dispatching algorithms with
detailed discussions. However, as our main purpose is to illustrate the impact of malware epidemic on distributed software systems, the algorithm for determining task dispatch is neglected.

In this section, it is assumed that the input task will be divided into two sub tasks. Detailed information of the extended target distributed system network is provided in the following tables. One sub-task will be processed by node 1 and 2 and the other sub-task will be processed by nodes 3, 4 and 5. The task requires 30 mega bits of data to be processed. 20 mega bits of data will be processed by node 1 and 2 and the other 10 mega bits will be processed by node 3, 4 and 5. It can be concluded from the following information that data exchange within nodes takes much more time than data processing and it would take at least 5 seconds (given that all communication channels are healthy) for data to be exchanged within different node sets while it will only take less than 0.4 seconds for data to be processed in each individual node (given that all nodes are healthy). To simplify the calculation process, we regard the service of distributed software to be reliable if the input task can be processed within 5.5 seconds.
The sub-task nodes sets are presented in Table 5.3 and the failure probability of the internal communication channels of the distributed system are presented in Table 5.4. Task spanning trees can be found in Figure 5.6. Since the network topology is not a complete graph, there are only 3 task spanning trees, namely $E_1 = \{2,3\}, E_2 = \{1,3\}, E_3 = \{1,5\}$.
Service Reliability without Malware Attack

If the distributed system is free of software virus, then equation (5.20) will be reduced to

\[
\Psi = \max_{1 \leq k \leq M} \tau_k = \max_{1 \leq k \leq M} \left( \frac{d_k}{\theta_i} + \frac{d_k}{s_i} \right) = \max_{1 \leq k \leq M} \left( \frac{d_k}{\alpha_i} + \frac{d_k}{s_i} \right) \quad (5.30)
\]

For each $\Psi$, it is easy to prove that it is less than 5.5 seconds. And the service reliability can be computed as

\[
R_s(t) = P(E_1) \times 1 + P(E_1, E_2) \times 1 + P(\bar{E}_1, \bar{E}_2, E_3) \times 1
\]
\[
= 0.99 \times 0.99 + 0.01 \times 0.01 \times 0.995 = 0.9999995 \quad (5.31)
\]
As can be concluded from the above calculations, the service of the 5-node distributed system is very reliable if it is not under malware attack and the service reliability is very stable and does not change with time. The only factor that affects the service reliability is the reliability of internal communication channels of the distributed system.

**Service Reliability with Malware Attack**

If the distributed system is under malware infection and node 1 has already been completely infected at the starting point, then the data processing time in different nodes need to be reconsidered. According to equation (5.20), the data processing time of $E_1$ can be written as

$$
\Psi_1(t) = \max_{1 \leq k \leq 2} \tau_k = \max_{1 \leq k \leq 2} \left( \frac{d_k}{\theta_i} + \frac{d_k}{s_i} \right)
$$

$$
= \max \left\{ \left( \frac{d_1}{\alpha_2(1-x_2(t))} + \frac{d_1}{s_1} \right), \left( \frac{d_2}{\alpha_3(1-x_3(t))} + \frac{d_2}{s_2} \right) \right\}
$$

(5.32)

where $x_i(t)$ follows a normal distribution $N(\mu_i(t), \sigma_i(t))$ which is discussed in Section 5.3.1. The probability that $\Psi_1(t) < T$ of the distributed system under virus infection can be derived as:

$$
\Pr\{\Psi_1(t) < T\} = \Pr\left\{ \frac{d_1}{\alpha_2(1-x_2(t))} + \frac{d_1}{s_1} \right\} \times \Pr\left\{ \frac{d_2}{\alpha_3(1-x_3(t))} + \frac{d_2}{s_2} \right\} < T
$$

$$
+ \Pr\left\{ \frac{d_1}{\alpha_2(1-x_2(t))} + \frac{d_1}{s_1} \right\} < \left( \frac{d_2}{\alpha_3(1-x_3(t))} + \frac{d_2}{s_2} \right) \times \Pr\left\{ \frac{d_2}{\alpha_3(1-x_3(t))} + \frac{d_2}{s_2} \right\} < T
$$

(5.33)
Since \( x_1(t) \) and \( x_5(t) \) both follow certain known normal distributions, the above probability equation can be solved numerically. Similarly, the probability of \( \text{Pr}\{ \Psi_2(t) < T_E \} \) and \( \text{Pr}\{ \Psi_3(t) < T_E \} \) can be obtained and the service reliability can be computed as:

\[
R_3(t) = P(E_1) \times \text{Pr}\{ \Psi_1(t) < T_E \} + P(\overline{E_1}E_2) \times \text{Pr}\{ \Psi_2(t) < T_E \} + P(\overline{E_1}\overline{E_2}E_3) \times \text{Pr}\{ \Psi_3(t) < T_E \}
\]

(5.34)

The service reliability over time is illustrated in the graph below.

*Figure 5-7 Service Reliability of the 5-node Distributed System over Time*
Figure 5.7 shows the impact of virus spreading on service reliability of the distributed system. An apparent deterioration of service reliability can be observed in Figure 5.7. As can be concluded from Figure 5-7, the service reliability of the distributed system continues degrading at the beginning. Such phenomenon can be attributed to the fact that infected nodes will keep spreading malware, which lowers the processing capability of the uninfected nodes. However, since all nodes are able to defend malware to some extent, the distributed system as a whole will not keep degrading all the time. When the malware defense rate catches up with the malware spread rate, the system will be in a relatively stable state. As a result, it is expected that the service reliability would also remain stable. Such a speculation is also validated in the above figure in the form of a stable curve at later times.
Chapter 6  Conclusions and Future Works

Aiming at proposing a proper framework for quantitatively estimating software quality, under which the problem of software maintenance, software structure and software security issues can be analyzed in an integrated manner, this thesis dedicates several studies on software availability, reliability and service reliability modeling by taking different external factors into consideration. From the perspectives of software engineering, these works provide both theoretical and practical advice for software practitioners in dealing with software maintenance, software architecture design and software security analyses problems. From the perspectives of industrial engineering, these works can be regarded as successful applications of IE methodologies such as stochastic modelling, probabilistic modelling and optimization. To highlight the contributions as well as limitations of the research in this thesis, we summarize the work done in each chapter, together with discussions of possible future research directions in the following paragraphs.

6.1 Conclusions and Contributions

In Chapter III, the problem of software availability modelling is investigated. The origin of software availability—software maintenance is first considered. A maintenance effort model is proposed to describe the maintenance activities during software maintenance phase and provides satisfactory results in field case validation. Based on the knowledge gained from the modeling of software maintenance process and previous literature review, a basic unified software availability model which is based on non-homogeneous Poisson process is then introduced and applied for determining the problem of optimal software maintenance policy from the availability and cost-effective criterion. The results show that our proposed model can not only quantitatively assess software availability with confidence but also can be used for
solving industrial problems in an integrated manner. In reality, the proposed software availability model can be used for estimating availability-sensitive software systems such as real time control system in chemical industry. On the other hand, our proposed model of searching for optimal software maintenance policy has greater application potentials. For example, in the manufacturing industry, where the cost of maintaining manufacturing execution software systems is high, our proposed model for optimal software maintenance policy can reduce this type of cost.

In Chapter IV, we shift our focus to the impact of fault-tolerance on software availability and the N-version programming technique is explicitly examined. Due to the difficulties of tractability of NHPP models, we proposed a Markov chain-based software model to analyze the availability behavior of N-version software systems. Results show clear evidence that fault-tolerance techniques have apparent positive impact on software availability and the interactions between different versions play the main role in the enhancement of software availability performance and our proposed model can quantitatively analyze these impacts in an integrated manner. A systematic approach is also proposed for choosing the optimal software structure when software practitioners are facing both availability and budget constraints. Similar to the availability model we proposed in Chapter III, the availability model proposed in this chapter is more specific and can be applied exclusively to N-version systems, such as financial clearance systems (Kan, 2003). By using the approaches presented in this chapter, decision makers would have a prior image of how the target software system behaves and how much the cost would be. And this helps a lot in facilitating the decision making process.

In Chapter V, the impact of malicious software is studied. Two virus epidemic models within distributed software systems are proposed, based on which meaningful quality metrics
such as software availability, software reliability and software service reliability can be derived. Continuum model and graph theory is applied in this study for the analysis of software quality degradation and we also proposed an algorithm, which is easy to be deployed on computers, to calculate the service reliability of a distributed software system under malware attack. The results show that malware have apparent negative impact on the quality of distributed software systems, however, the service reliability is more vulnerable to malware attacks when compared to the other two metrics. The contribution of this piece of work is that it is the first piece of research in the field of software quality that systematically analyzes the impact of malware. With the proposed models in this chapter, readers can quantitatively estimate the quality degradation of malware infection. What is more, this chapter emphasized on distributed software system, which is a special case of the grid system or cloud system. This also indicates the current work can be extended to describe similar symptoms of grid system or cloud system. On the other hand, the graph modeling of distributed software can also be used to analyze any general computer networks, such as the Local Area Network (LAN) or even the internet.

6.2 Directions of Future Research

In Chapter III, for the purpose of clarity and simplification, we made several strong assumptions when building the proposed software availability model, one of which is perfect maintenance assumption. In fact, software maintenance must be performed by human beings, which are fault-prone creatures and the assumption of perfect maintenance is thus hard to validate in real industrial cases. Therefore, extending the basic software availability model to incorporate imperfect maintenance is of future research interest. In addition, only one set of real industrial dataset is used for experiments in this study while more analysis depends on simulation.
The future work should also include more analysis of real world datasets. Another possible future direction is not closely related to the availability modeling but to the modeling of software maintenance. When we were investigating the problem of software maintenance effort, we found that there were several different metrics that could be used as indicators of maintenance effort, such as money, changes in lines of source code, man-hour records and so on. We believe that if we can propose a possible approach to take all these types of metrics into consideration, then the maintenance process can be better described. A compound NHPP process modelling would be a possible approach for such a further analysis.

In Chapter IV, the availability problem of N-version software system is of concern and there are also limitations in this study. Firstly, the failure correlation, which commonly exists in fault-tolerant systems, is not considered. Only by taking the problem of failure correlation into consideration can our model better present the reality. Secondly, we ignored the complexity of the voting mechanisms within N-version systems and assumed the results depend on the output of a majority vote. In fact, voters in N-version systems are complex and also affect software quality. The future work should conduct a deep investigation into the interactions between the output of each version and the voter. The last but not the least, this study still heavily depends on simulation data and future work should focus more on field cases.

In Chapter V, a quality analysis of distributed software considering malware attack is presented. The limitations in this study are also listed. In the virus epidemic model, the factor of virus defense is considered and it is not properly modeled but just regarded as some known and constant parameter. In fact, virus defense ability depends on each individual node within the distributed system and can be upgraded if necessary. Such upgrade is costly and need to be carefully designed. On the other hand, we also noticed in our investigation that certain
communication channels have larger weight than the others, which makes the node that are
c connected by these communication channels more vulnerable to malware attack. We believe that
in the graph representation of distributed software or even of any general forms of computer
network, there is a certain type of “critical information/virus epidemic path”. This type of critical
path is of great theoretical and practical value because if we can find this type of path, then we
can defend the entire system with minimal cost by defending the nodes within this path. With
proper mathematical formulation and modeling, we believe that a general algorithm of searching
for the critical path can be obtained. For future research, work can be done in searching for the
optimal malware defense strategy by considering the critical virus epidemic path and the cost of
enhancing virus defense abilities of each individual node.
Bibliography


168
Jacoby R., Tohma Y. (1991). 'Parameter value computation by least square method and evaluation of software availability and reliability at service-operation by the hyper-geometric distribution software reliability growth model (HGDM)', Proc. the 13th international conference on Software engineering, Austin, Texas, USA, pp.226-237


Bibliography


174

176

Appendix  Service Reliability Calculation Algorithm

Begin

\[ R_s(t) \leftarrow P(E_1)P(\Psi_1 \leq T_E) \]  
\[ \text{\textit{\small{\# initialize the system reliability, where } P(E_1) \text{ is the}}\]  
\[ \text{product of all links' functioning probabilities in the 1st MTST:} \]  
\[ P(E) = \prod_{i \in \Lambda_s} p_i^1, \text{ and } P(\Psi_1 \leq T_E) \text{ can be obtained by directly} \]  
\[ \text{using equation (5.29).} \]

\[ \Theta \leftarrow \{\emptyset\} \]  
\[ \text{\textit{\small{\# initialize } } \Theta \text{ which will be used to store the union of the} \]  
\[ \text{events } (E). \]

For all \( \Lambda_{h^*} \in \Lambda_s (1 < h^* \leq H) \) do

\[ \Theta \leftarrow [E_1 \cup \cdots \cup E_{h^*-1}] \]  
\[ \text{\textit{\small{\# let } } \Theta \text{ store the union of the events } (E) \text{ that are the} \]  
\[ \text{predecessors of the current event } E_{h^*}. \]

\[ F \leftarrow \text{Reduction}(\Theta, E_{h^*}) \]  
\[ \text{\textit{\small{\# } } F \text{ is the union of all preceding events (before } E_{h^*} \text{)} in} \]  
\[ \text{which any link that is present in both } E_{h^*} \text{ and any of its} \]  
\[ \text{preceeding events is deleted from those preceding events.} \]

\[ S \leftarrow \text{Disjoint}(F) \]  
\[ \text{\textit{\small{\# } } S \text{ is the sum of disjoint products representing } \overline{F}. \text{ This} \]  
\[ \text{function is used to decomposes } \overline{F} \text{ into } S \text{ to compute the} \]  
\[ \text{probability of } \overline{F}. \text{ As } F \text{ and } E_{h^*} \text{ have no comment elements,} \]  
\[ \text{the probability can be written as} \]
\[ P(\bigcap_{i=1}^{n} E_i \cap \bigcap_{h=1}^{r} E_{h^*}) = P(\overline{F} \cap E_{h^*}) = P(\overline{F})P(E_{h^*}) \]

\[ R_s(t) \leftarrow R_s(t) + P(E_{h^v})P(S)P(\Psi_{h^v} \leq T_E) \]

End do

End

\textbf{Function} Reduction(\Theta, E_{h^v})  \\
\text{\texttt{Il}} this function detects the links that appear in both \Theta  \\
and \( E_{h^v} \), and deletes those links from set \Theta

Begin

for all \( l_i^\Theta \in \Lambda_\Theta (1 \leq i \leq n_\Theta) \) do  \\
\text{\texttt{Il}} where \( n_\Theta \) is the total number of links involved in the events  \\
union: \Theta

for all \( l_j^{h^v} \in \Lambda_{h^v} (1 \leq j \leq n_{h^v}) \) do  \\
\text{\texttt{Il}} where \( n_{h^v} \) is the total number of links involved in the event:  \\
\( E_{h^v} \)

if \( l_i^\Theta = l_j^{h^v} \) then \( e_i^\Theta = 1 \)  \\
\text{\texttt{Il}} to delete the links that appear in both \( \Theta \) and \( E_{h^v} \) from  \\
set \( \Theta \), assign Boolean value of ‘1’ to those links

end do

end do

\( F \leftarrow \Theta \)  \\
\text{\texttt{Il}} simplify the event union \( \Theta \) with the Boolean values ‘1’  \\
and assign it to \( F \)
Function Disjoint($F$)

//this function decomposes the event $F$ into a sum of disjoint products. It is noted that $F$ has the following expression:

$$F = E_1^{(r)} \cup E_2^{(r)} \cup \ldots \cup E_{n-1}^{(r)} = \overline{E_1^{(r)}} \cap E_2^{(r)} \cap \ldots \cap E_{n-1}^{(r)} = \overline{E_1^{(r)}} \cap \ldots \cap \overline{E_{n-1}^{(r)}}$$

where $E_i^{(r)}$ is the reduced version of event $E_i$ (by applying the reduction function), and $\overline{\overline{E}}$ is the exclusive operator.

Begin

$S \leftarrow \overline{\overline{E_1^{(r)}}}$

// $S$ is the set that stores the sum of disjoint products. This step initializes $S$. By the operation rule of exclusive operator, we have

$$\overline{\overline{E_1^{(r)}}} = \overline{\overline{E_1^{(r)}}} = e_1^1 \cup e_1^2 \cup \ldots \cup e_1^{r-1} \cup e_1^n$$

where $n$ is the total number of links in the reduced event $E_1^{(r)}$.

for all $2 \leq i \leq h^* - 1$

// where $(h^* - 1)$ is the total number of reduced events in the events union: $F$.

$S \leftarrow S \cap \overline{\overline{E_i^{(r)}}}$

// this step applies the operator $\overline{\overline{E}}$ to decomposes $E_i^{(r)}$ into a sum of disjoint products. For instance,

$$\overline{\overline{E_i^{(r)}}} = \overline{\overline{E_i^{(r)}}} = e_i^1 \cup e_i^2 \cup \ldots \cup e_i^{r-1} \cup e_i^n$$

where $n$ is the total number of links in the reduced event $E_i^{(r)}$.

return $F$

end function
End do

Return $S$ \hspace{1cm} //expands $S$ into the form of unions (e.g. $S = (e_1 \ldots e_n) \cup (e_{m+1} \ldots e_m)$ by the distributive law.

End function