Architecture-Based Software Reliability Modeling

by

Wen-Li Wang

A Dissertation

Submitted to the University at Albany, State University of New York

in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

College of Arts & Sciences

Department of Computer Science

2002

Architecture-Based Software Reliability Modeling

by

Wen-Li Wang

Abstract

Software failure, especially for large-scale software, can lead to a big catastrophe in many application domains. Therefore, reducing the chances of software failure and assuring software quality is highly desirable. Software reliability, a measure that estimates the probability that software will not cause the failure of a system for a specified time under specified conditions, thus is one of the key quality attributes. With the increasing sophistication of software infrastructures, the objective of this thesis is to model the reliability of large-scale software taking into account the architectural characteristics and complexity.

Traditional reliability models and approaches are subject to address either no system structures using a black-box approach or only simple homogeneous architectures using a white-box approach. Therefore, we proposed an architecture-based model, in which system structures are classified into architectural styles, in order to model the heterogeneity of different configurations and constraints. Our model extends the traditional Markov-based reliability models, a white-box approach, to ease the modeling of the interrelationships among components, and to assist in prioritizing the component improvement sequence.

However, the extension of traditional Markov-based models limits the modeling to only history-independent stochastic software behaviors: a problem resulting from the assumption that a software process follows a Markov process. Our model removes this fundamental barrier to address execution history, and both probabilistic and deterministic software behaviors, utilizing grammar rules and the structures of binomial trees without the necessity of history keeping. The grammar rules tackle the execution dependencies among components, while the structures of binomial trees facilitate the modeling of all possible execution paths.

Our architecture-based reliability model, taking the styles and grammar as inputs, not only addresses modern software infrastructures, but also resolves the traditional modeling limitations. Because of a white-box approach, this model facilitates making effective design decisions and choosing suitable software components at an early stage of the software development process. It eliminates the need for retesting the whole software system, once configurations are changed. Therefore, our architecture-based reliability model is expected to gain a much wider scope of applications.

Acknowledgements

I would like to sincerely thank my advisor, Prof. Mei-Hwa Chen for her guidance, assistance, and encouragement in completing this thesis. She taught me how to conduct research and resolve problems in a systematic way. She triggered my research interests and helped me go through each tough time. She insisted on accuracy with precise details and applicability to industry, from which I learned to make my research more complete. Her patience and kindness lifted me up when I was down. Her ability and enthusiasm in research strongly stimulate me and amaze me.

I would like to thank the members of my committee – Prof. Daniel J. Rosenkrantz and Prof. Seth Chaiken. Professor Rosenkrantz gave me much appreciated encouragement and invaluable suggestions. He patiently discussed with my research directions and pointed out the potential difficulties of my research work. His rigor and precision helped me to face tough research challenges. I am also grateful to Professor Chaiken for his questionings, which helped me to build my research foundations.

I would also like to thank Prof. S. S. Ravi for his assistance. He patiently listened to my modeling ideas and helped me understand the complexity of algorithms. His detailed illustrations in computing stimulate not only my research interests, but also my teaching interests.

I am thankful to my friend Lance Nevard for his humor, and his editing of my thesis. Mr. Nevard has helped me go through several of my papers and corrected the ambiguities in my writings. I also want to thank Pat Keller for being my friend and helping me with a myriad of administrative details.

Last but not the least, I thank my wife, Mei-Huei Tang, for not just supporting me for the past several years, but also patiently listening to my crazy ideas. Her sacrifice in taking care of our lovely baby helped me to complete this thesis smoothly. I also want to thank my parents for continually supporting my progress and encouraging me not to quit.

I am afraid that I am leaving out many people that have helped me over the years. I want to thank all of them and feel sorry that I cannot acknowledge each one of them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"*It's fine to celebrate success but it is more important to heed the lessons of failure*", stated Bill Gates (The Guardian, 27 April 1995). Software failure can lead to a big catastrophe in many application domains, such as aviation, business, finance, and military actions, etc. Take the aerospace domain for example. NASA's 1999 Mars Exploration Program suffered a $125 million loss when the thrusters on the Mars Polar Lander shutdown above the surface causing the vehicle to crash. Although the crash was attributed to human communication errors, software failures occurred when two microprobes of Deep Space 2, which should have been released from the Lander before the crash, were never activated.

Software failure is "the inability of a system or system component to perform a required function within specified limits" [35]. Although some factors to software failures are related to management errors, many are closely related to software complexity and its reliability. In [22], several software failure stories were discussed regarding poor software qualities that fail to carry out the required functions. The following three cases demonstrate that software failures may not only cause huge economic loss, but can also jeopardize human lives. Nevertheless, with early and reasonable investment on maintaining good software quality, the chance of software failures is very likely reducible.

- The first case is one of the most costly commercial systems "Confirm project", which resulted in a $213 million loss. This project, based on the successful story of American Airlines' SABRE computer reservation system, started out as a joint venture but ended up as a disaster. The major problems with Confirm were associated with the operation of the TMF sub-system, a bridge between two mainframes. This TMF sub-system was insufficient to handle the complexity of providing an application-to-application link between the mainframe processors for sixty applications. Essentially, the TMF did not work correctly and the resulting database was apparently irrecoverable after a crash.

- The second case depicts that an early corrective action is possible to prevent a big loss caused by software failure. The show business based Performing Right Society (PRS) in London failed in developing their PROMS project, which was intended to collect intellectual property royalties for copyright holders. The cost of software development was over $12 million, and this big loss due to software failure was very likely preventable by taking a cheaper $0.3 million corrective action at an early stage of the software development life cycle to reveal the possible faults.

- The third case illustrates the importance of software quality, where low quality software delays the processing or response to emergencies; thus puts human lives at risk. The big disaster was the London Ambulance Service (LAS) Computer

Aided Dispatch system. The system design was for normal operations but could not address imperfect situations. As a result, ambulances failed to arrive at an incident, arrived late, or turned up two or more at a time.

Software engineering aims at ensuring software quality to meet user needs. To evaluate the quality of large-scale software, the emphasis is on how good the components of a software system can incorporate correctly and generate expected outcome. Therefore, we study configurations of software components and their impact on overall system reliability. The configurations can be different types of system structures, in order to satisfy user requirements. If the reliability or quality of software cannot achieve a desired level, it is necessary to invest more labor to make further improvements, such as more testing efforts, better document keeping, or new component upgrades. Although the efforts to enhance software reliability or quality can be expensive, however like the PROMS project, investing 0.3 million of corrective action may save 12 million dollars loss from software failure, and successful software can very possibly bring in significant profits. Therefore, the investment in software reliability enhancement can be extremely fruitful.

Software reliability is often defined as "the probability that software will not cause the failure of a system for a specified time under specified conditions" [35]. This type of measurement is sensitive to time. When time is not an issue, software reliability is also accepted as the probability of failure-free operation of a computer program for a typical set of user inputs in a specified environment [14]. This typical set of user inputs exercise the possible software processes. Other popular measures concerning software reliability, such as Mean Time To Failure (MTTF) and failure intensity, which is the expected number of failures per unit time interval, are also frequently used. To assure high quality software, software reliability measures are thus utilized as one of the important quality attributes to evaluate software quality.

For more than three decades, software reliability modeling has been studied and a number of analytical models and simulation techniques were proposed for estimating the reliability of software systems. These existing approaches can be roughly classified into three categories. The first category is a time domain approach, also called software reliability growth models (SRGMs) [21,58]. SRGMs are a statistical approach, which has the form of a random process that describes the behavior of failures with time. SRGMs are a black-box approach, which is not concerned with system structures. These models utilize test data to estimate software reliability, and assume testing is conducted using random testing. The test cases are created based on an operational profile. In order to apply SRGMs, the distribution function for the random process, which describes the behavior of failures with time, is first observed from the test results. Based on this distribution function, software reliability is computed accordingly. Therefore, SRGMs rely on test data and the distribution function decides the models for software reliability measurement.

Unlike SRGMs, the second category is discrete time or discrete event Markov-based reliability models, using a white-box approach. These models are concerned with system structures and are capable of addressing certain interactions and interrelationships between two software modules. The discrete time Markov-based reliability models are suitable for modeling simple homogeneous software, such as software with sequential execution, branching, and looping system structures. However, software can have more sophisticated infrastructures, such as parallel computation to enhance performance, fault tolerance to improve reliability, and various interactions and intercommunications, etc. In order to easily utilize the discrete time Markov models, traditional modeling techniques assumed that a software process follows a Markov process, in which the next software module is affected by the current module only regardless of who were the ancestors. Unfortunately, this assumption limits the modeling of complex system structures, heterogeneous architectures, and history-dependent execution paths.

Beside these two types of analytical models, the third category is based on simulations or experiments. Discrete-event simulation approaches were proposed to simulate software behaviors through predefined discrete events, where the transition from one step to another is based on a triggered event and nothing occurs between events. Experiments were also conducted to estimate software reliability using software components. The system reliability is the average over all test runs. However, simulation or experimental approaches are very time-consuming with narrower application domains, and the estimation results may vary due to data variation caused by random variables.

Despite the available approaches and models, we develop an architecture-based software reliability model. We define our reliability measure as the probability of failure-free operation of an overall software architecture for a typical set of component interactions in a specified environment. This typical set of component interactions is used to exercise possible execution paths.

Generally speaking, the bigger the disaster caused by software failures is commonly the larger the software systems. With the increasing complexity of software infrastructures, our primary objective is to model large-scale software, taking into account the architectural characteristics and complexity. Typically, large-scale software is complex, and composed of a variety of components, and interfaces. The components and interfaces can form different configurations and each configuration can have different topologies and constraints. This results in heterogeneous system structures. At the design phase of large-scale software, the major concerns are not data structures or algorithms, but the overall system structures. Therefore, how to configure the global system, decompose interacting subsystems, designate component functions, classify data communication protocols, evaluate complete system, and select a better-fit design are the major issues of software architecture. Because changing the architectural designs at a later phase of the software life cycle can be difficult and expensive, a correct decision needs to be made in the early design phase of the software life cycle.

Therefore, the second objective of our model is to support decision-making in the choosing of a better-fit design, early in the software process. Since SRGMs rely on the test data or the distribution of the failure process, they are not suitable for such an early prediction purpose. On the other hand, the traditional discrete time Markov-based reliability models are a white-box approach that can support this goal of providing a *relative* analysis result. Unfortunately, the traditional models can only model simple system structures so that if systems have complex design structures, the modeling will be difficult.

The third objective is to accommodate frequent component upgrades or updates, due to easy component plug-and-play. With the prevalence of today's component-based software, software components can be developed separately, remotely, and by different groups. The interfaces for interactions and intercommunications are well defined in order for components to communicate with each other. In addition, the commercial-off-the-shelf software (COTS) further facilitates the ease of acquiring new software components for use. Therefore, the locus of adapting frequent component upgrades or updates is to configure the newly available components with the existing components, realize the limitations of their configurations, and define the interface protocols among the new set of components. To accommodate the frequent component upgrade or update, the black-box approach of SRGMs will require the retesting of the whole program, which is undesirable for component-based software reliability measurement.

To achieve these three objectives, we introduce a two-phase approach for developing our architecture-based software reliability model.

- Phase one makes use of architectural styles to identify system structures in a formal way and resolves the modeling limitations of homogeneous Markov-based reliability models to address heterogeneous software behaviors. The architectural styles serve as high-level design patterns, which regulate the configurations and constraints. A new state model is developed to model both homogeneous and heterogeneous characteristics of architectural styles, in order to take into account more complex system structures, such as parallel computing, client-server architecture, and fault tolerance, etc.

- The second phase is to remove the fundamental barrier of traditional Markov-based reliability models that assume a software process follows a Markov process. A Markov process implies that the next state depends probabilistically on the present state only and is independent of execution history. Although a Markov model is a *stochastic* model, good for modeling uncertainties, traditional Markov-based models are incapable of addressing execution history and non-probabilistic system behaviors. Phase two significantly broadens the application domains by enabling reliability measurement on software that does not have Markov properties.

Our architecture-based software reliability model, a white-box approach based on the discrete-time Markov models, can provide a guideline for choosing a relatively better-fit architecture design by changing and exercising different architectural styles into a system. This supports the decision making for choosing alternative designs or components to meet reliability requirements and to lower the chance of future software failure. The model overcomes the obstacles of traditional approaches to modeling complex system structures as well as the heterogeneity of large-scale software. By utilizing architectural styles and extending traditional discrete time Markov-based models, our architecture-based reliability model can address both homogeneous and heterogeneous component interactions and interrelationships, and examine the critical components to the overall system reliability. By removing the fundamental barrier, the model can address execution history as well as both probabilistic and deterministic system behaviors. This not only indicates to us which architecture to pursue, but also prioritizes the component improvement sequence. As improving software reliability improves software quality, we thus reduce the chance of software failure.

In this thesis, Chapter II presents the literature reviews on existing software reliability models and approaches, and makes sense of the inspiration, evolution and significance of architecture design as well as its position in the software development life cycle. In Chapter III, we will discuss the utilization of Markov models to build the foundation of our architecture-based reliability model. Chapter IV demonstrates our probabilistic architecture-based reliability model, which is able to take into account software architecture and uncertain component transitions. Chapter V will introduce history-dependent deterministic software reliability modeling to resolve the limitations of the probabilistic architecture-based reliability model. Chapter VI draws the conclusions.

Chapter 2

Literature Review

2.1 Software Reliability Modeling

Software reliability modeling has been studied for more than three decades. To measure software reliability, the existing approaches can be roughly classified into these three categories: *software reliability growth models* (*SRGMs*), *Markov-based reliability models*, and *simulations and experiments*.

2.1.1 Software Reliability Growth Models

Software reliability growth models usually address the failure behaviors of a software system as random processes. The models cover software either with repairs, or without repairs, and commonly assume that failures are independent of each other. Here presents the historical development of software reliability growth models [21,58]:

Hudson [34] conducted the first study of software reliability. He viewed program errors as a birth or death process. A fault generation is a birth, while a fault correction is a death. The transition probabilities are related to the birth and death functions. He confined his work to pure death processes and assumed that the rate of fault detection would increase with time. It was shown that the number of faults detected follows a binomial distribution.

Jelinski and Moranda [38] as well as Shooman [73] made another major step. They assumed a hazard rate for failures was constant and proportional to the number of faults remaining. The hazard rate changes at each fault correction by a constant amount, but is constant between corrections. Moranda [56] later proposed two variants of the Jelinski-Moranda model. The first has a hazard rate decrease in steps that form a geometric progression. The second further has the decrements occur at fixed intervals rather than at each failure correction.

Schick and Wolverton [64] proposed another model, assuming that the hazard rate was proportional to the product of the number of faults remaining and the time. Therefore, the size of the changes in hazard rate increases with time. Wagoner [84] had a slightly different assumption that the hazard rate was proportional to the product of the number of faults remaining and a power of the time. The power can be varied to fit the data.

Schneidewind [66] approached software reliability modeling starting from an empirical point of view. He found that the best distribution for reliability measurement varied from project to project. Therefore, he suggested not using point estimates but the confidence intervals for the parameters. In [67], he viewed fault detections per time interval as a non-homogeneous Poisson process, a non-linear function of time, with an exponential mean value function.

Musa [57] presented an execution time model of software reliability that broke new ground in several ways. He claimed that software reliability measurement should be based on processor execution time not the calendar time. He observed that when failure rates were taken based on execution time, the fault correction rate was typically proportional to the hazard rate. The result is the aforementioned variability, noted by Schneidewind, from project to project did not occur. This approach became universal and easier to apply.

Littlewood and Verrall [47] proposed a Bayesian approach to software reliability measurement. They modeled the hazard rate as a random variable instead of a function of the number of faults remaining. Therefore, software reliability is viewed as a measure of strength of belief that a program will function successfully, which contrasts with the classical view of reliability as the number of successful executions out of the total number of executions. The concept of the hazard rate as a random variable can characterize reliability change.

Littlewood [50] later proposed a different fault model, a variant of the general Littlewood-Verrall model. The hazard is still viewed as a random variable with one hypothesis that failures occur with different frequencies. Typically, faults that occur most frequently will be detected and corrected first. He considered that uncertainties in reliability growth might result more from uncertainties in the relative frequencies of execution of different input states than uncertainties in fault correction.

Goel and Okumoto [28], based on the assumptions of those of Jelinski and Moranda, described failure detection as a non-homogeneous Poisson process (NHPP) with an exponentially decaying rate function. Both the cumulative number of detected failures and the number of remaining failures are Poisson processes. Yamada, Ohba, and Osaki [86] modified the cumulative number of detected failures in the NHPP model to an *S*-shaped curve. In addition, Goel and Okumoto [29] also developed a model for imperfect debugging from the modification of Jelinski-Moranda model. They viewed debugging as a Markov process, with appropriate transition probabilities among states. Kremer [41] extended this model, accepting the idea that the repair activity has the possibility of introducing new faults.

Before late 1970s, most of the studies are looking for different modeling possibilities. In the early 1980s, the focus moved to the comparison of software reliability models to choose better ones. Iannino et al. [36] worked out a consensus on the comparison criteria. Musa and Okumoto [59] clarified and organized comparisons, and suggested possible new models. The following shows the classification scheme developed by Musa, Iannino, and Okumoto [58] for software reliability models, which are classified in terms of five different attributes:

1. *time domain*: calendar time or execution (CPU or processor) time.
2. *category*: the number of failures that can be experienced in infinite time is finite or infinite.
3. *type*: the distribution of the number of failures experienced by time t.
4. *class* (finite failures category only): functional form of the failure intensity in terms of time, and
5. *family* (infinite failures category only): functional form of the failure intensity in terms of the expected number of failures experienced.

Based on the above classification scheme, software reliability models can be classified into the following groups:

1. *Exponential Failure Time Class of Models*, including Jelinski-Moranda model [38], non-homogeneous Poisson process (NHPP) model [29], Schneidewind's model [67], Musa's basic execution time model [57,58], and hyper-exponential model [44,87].

2. *Weibull and Gamma Failure Time Class of Models*, including binomial type Weibull model [64], and Yamada's *S*-shaped reliability growth model [86].

3. *Infinite Failure Category Models*, including Duane's model [18], Moranda's Geometric model, and Musa-Okumoto logarithmic Poisson [60].

4. *Bayesian Models*, including Littlewood-Verrall reliability growth model [47], Kyparisis and Singpurwalla's Bayesian non-homogeneous Poisson process model [43], and Liu's Bayesian geometric models [51], etc.

The classification and comparisons lead to the development of Musa-Okumoto logarithmic Poisson execution time model [60] with simplicity and good predictive validity. This model is based on a non-homogeneous Poisson process with a logarithmic failure intensity function. This logarithmic function decreases exponentially with expected failures experienced, so that the failures repaired in an early stage reduce the failure intensity more significantly than those in a late stage.

Software Reliability Growth Models (SRGMs) are a time domain model, employing test history to predict *mean time to failure* (MTTF) or *the probability of failure free operation for a specified period of time* for software [21,58]. SRGMs assume testing is conducted using random testing, and the test cases are created based on an operational profile. Using a black-box approach, these models are not concerned with system structures. Therefore, if a change or an update occurs to the structure or software components, the models require re-testing of the whole software system. With the characteristics of easy component plug-and-play, upgrade, or update, software components ease the software development of large-scale software, but hinder the use of SRGMs due to the repeated testing efforts.

2.1.2 Markov-Based Reliability Models

A stochastic process is a collection of random variables. If each variable represent the state of a system at some specific index, there is a conditional probability of being in each state. A finite-state Markov process is a stochastic process withholding the following conditions.

1. There is a finite set of states.
2. The transition to the next state will depend probabilistically on the present state alone and is independent of execution history.
3. The transition probability from one state to another does not change over time.
4. A set of initial probabilities is defined for all states.

Markov models can be classified based on model types or model applications. By model types, a Markov model can be either a discrete Markov model or a continuous

Markov model. Both discrete and continuous Markov models can be either homogeneous or non-homogeneous. Homogeneous model refers to constant or time invariant transition rates. Non-homogeneous model on the other hand refers to time variant transition rates. If a Markov model is continuous time, the transitions can occur at any instance of time and the transition probabilities follow a continuous distribution. If a Markov model is discrete time or discrete event, the transitions occur only at discrete intervals of time or at discrete events, and the transition probabilities follow a discrete distribution.

By model applications, the application of a Markov model can be categorized into repairable systems or non-repairable systems. A repairable system is continuously available for repair, while a non-repairable system is not available for repair once in the operation. Repairable systems contain cycles to restore a previous state based on a repair rate, which can be a function of software complexity, or other variables. The repair rate can be fixed or vary with time. For repairable systems, both reliability and availability can be computed. Non-repairable systems have no cycles to traverse back to a state previously visited. For such systems, reliability or reliability for periodically renewed system can be computed.

Furthermore, there is another type of model, the semi-Markov model, which is closely related to the Markov model. The semi-Markov model [62] is basically like the Markov model from the state transitions point of view. Unlike the Markov model, the semi-Markov model extends the usual discrete-time Markov model by incorporating a continuous model of time. There is a random or constant amount of time between state changes, and this local holding time can be any distribution. Therefore, a process can be held in a state for a positive amount of time by a holding probability distribution function.

Using semi-Markov models thus can facilitate the production of the effect of competing events in fault handling, which is contrast to instantaneous coverage that does not.

Most of the reliability models discussed in Section 2.1.1 can be described in terms of a continuous-time Markov process, except for the geometric model of Moranda [56] and the Bayesian model of Littlewood and Verrall [47]. Markov processes are useful in modeling random behavior of software in time, such as faults remaining at time *t* and failures experienced by time *t*.

From a number of studies [14,31,44,48,85] and tools [39] adopting Markov models to measure the reliability of modular software, Cheung [14] proposed a user-oriented reliability model to measure the reliability of service that a system provides to a user community. A discrete Markov model was formulated based on the knowledge of individual module reliability and inter-module transition probabilities. Sensitivity analysis was also conducted to determine modules most critical to system reliability. In this model, module behaviors, including the sequential, branching, and cyclic executions, were taken into account. In Littlewood's reliability model [48], a modular program is treated as transfers of control between modules following a semi-Markov process. Each module is failure-prone, and different failure processes are assumed to be Poisson distribution. The KAT Approach, proposed by Laprie et. al [44], was developed for modeling and evaluating the reliability and availability of multi-component systems from the knowledge of the reliability growth of their components. A knowledge-action transformation approach was presented to account for reliability growth phenomena, which enables the estimation and prediction of the reliability and availability of multi-component systems. The transformed action model is a transformation of classical

Markov models into other Markov models that account for reliability growth. This model can take simple system structures into account but the state expansion always goes up to non-polynomial number of states for an *n*-component system.

Markov-based reliability models assume that the control transfers among software components follow a Markov process. This implies that the future of the process depends only on the present state and is independent of execution history. Therefore, if a software process is assumed be a Markov process, the reliability models may still be able to handle software with simple homogeneous sequential execution and branching transitions, but are insufficient to model sophisticated structures and heterogeneous architectures.

2.1.3 Simulations and Experiments

In addition to analytical models, simulations and experiments were developed to predict and measure software reliability. Gokhale et al [30] proposed a discrete-event simulation to capture a detailed system structure and to study the influence of separate factors in a combined fashion on dependability measures. Krishnamurthy and Mathur [42] conducted an experiment to evaluate a method, *Component Based Reliability Estimation* (CBRE), to estimate software reliability using software components. CBRE involves computing path reliability estimates based on the sequence of components executed for each test input and the system reliability is the average over all test runs. Li et al [46] presented a methodology and accompanying toolset, W2S, for generating a simulator from a semi-formal architecture description, which allows an analysis of the system's reliability based on it's simulated behavior and performance. Gokhale et al [31] predicted architecture-based software reliability using a testing-based approach, which parameterized the analytic model of the software using measurements obtained from the regression test suite, and coverage measurements.

The above approaches are able to model certain system structures, but are usually expensive and time-consuming especially when applying to structure changes, and the application domains can be limited.

2.2 Overview of Software Architectures

Large-scale software systems particularly bring the revival interests in high-level design and drive the software architecture renaissance. Generally speaking, software architecture describes the organization or configuration of the overall system. Architectural decisions of software are usually made at the early stage of the software development life cycle. Typically, it is very difficult and expensive to change the decisions at the later phase.

There have been many topics dedicated to the software architectural level of design including *module interconnection languages (MILs), architectural styles identification and description, formal specification models of component integration mechanisms, architectural description languages (ADLs), and frameworks for domain-specific systems.*

The first *MIL*, proposed by DeRemer and Kron [17] in 1975, was designed to support the connection effort between modules. They argued that the modules creation and modules interconnection were different structural design efforts that could be designed independently. In general, flexible and high-level connections between subsystems help to build composable systems. However, *MILs* only support low-level interactions and can not reuse patterns of compositions.

Garlan and Shaw [27] elaborate the taxonomy of the broadly used patterns or idioms that emerge as some common *architectural styles.* An architectural style constrains both the system components and the formal relationships among the system components. It also includes the topological constraints on architectural descriptions. Since a style defines and limits the kinds of design components and their formal relationships, the architectural styles make easy understanding a system's configuration, and also permit specialized, style-specific analysis, etc.

Formal specification models are designed to solve the problems of many informal approaches. Inverardi and Wolf [37] brought up an idea using the chemical abstract machine model as the formal model for formal specification and analysis of software architectures. The architectural elements are represented as "molecules", and the floating molecules can only interact according to explicitly stated reaction rules. Abowd, Allen, and Garlan [3] choose different formal approach by taking architectural styles as an interpretation from syntax to semantics and outline a framework not only allowing the analysis within different architectural styles, but also providing a template for prescribing new architectural styles. Such formalism can assist in describing and analyzing software systems.

Architectural description languages (*ADLs*) compensate the inadequacy for architectural descriptions of today's programming languages by utilizing better notations to describe high-level views of component relationships, as well as their combinations and interactions.

For the *domain-specific software architectures* (*DSSAs*), there exists an idea that a common architecture can be extracted from a collection of related systems, thus new systems related to the specific problem domains can be built by using the common shared architectures. Such an idea may reduce the time and cost of producing specific application within a supported domain where frameworks are already available to increase product quality, improve manageability, and support software reuse, etc. There have been numerous industrial and defense research projects dedicated to creating domain-specific architectural styles or reference architectures for specific problem domains.

Software components, architectural styles, formal specification models, *ADLs*, and *DSSAs* are increasingly important to the emerging field of architectural level of design. They can provide a different meaning and utilization value for various sorts of people, based on their concerns. Typically, customers care about budget estimation, risk assessment, and progress tracking in order to reduce cost. Users care about requirement consistency, future needs accommodation, performance, and inter-operability. Developers expect sufficient details for design, references for selecting and assembling components, and interoperability maintenance of existing systems. Software engineers are concerned about requirements traceability, software reliability, trade-off analysis, and consistency of architecture. For software maintainers, they care about guidance on software modification, and guidance on architecture evolution. With the flexibility of component plug-and-play, the regulations of architectural styles, the verifications of formal models, the descriptions and prototyping of *ADLs*, and the reusability of *DSSAs*, the goal of software architecture to provide multiple views [55,61] to satisfy different individuals with different concerns can be fulfilled.

2.2.1 Definitions of Software Architecture

Garlan and Shaw [27] define software architecture as composed of *components, connectors, and configurations*. *Components* define the locus of computation. *Connectors* define the interactions (e.g. procedure call, data flow, implicit triggering, message passing, shared data, and instantiation [71]) between components. *Configurations* define the topology of the components and connectors.

From Perry and Wolf [61], software architecture is defined as a set of *architectural elements* that have a particular *form*, and an underlying *rationale*. *Architectural elements* include processing, data, and connecting elements. *Form* consists of weighted properties constraining the choice of architectural elements in addition to weighted relationships that constrain the interaction and organization of different elements in the architecture. *Rationale* captures the motivation for the choice of architectural style, elements, and the form. An architectural style constrains both the system components and the formal relationships among the system components. It also includes the topological constraints on architectural descriptions.

Jones [40] describes software architecture as a structure composed of components, and rules characterizing the interaction of these components.

2.2.2 Inspiration of Higher-Level Design

For large-scale systems, the major subject is usually subsystems and their interactions. This level of organization is the *software architectural level*, a high-level architecture design stage. *Abstractions* are the techniques to implement high-level components and to catch the intrinsic properties of subsystems and their interactions.

The essence of abstraction is to first recognize a *pattern*. Patterns are the way of capturing software developers' experiences and then communicating this information to the programmers. Abstractions can be used to identify recurring situations and suggest "which decisions to make, when and how to make them, and how they are the right decisions" [19]. Patterns also have strength for future reuse as well as to prevent the waste of time and energy, reinventing development processes to solve old problems.

After the recognition of a pattern, another aspect of abstraction is naming and defining the pattern, analyzing it, finding ways to specify it, and providing some way to invoke the pattern, by its name, without error-prone manual intervention [68]. This kind of process has some benefits, for it covers the implementation details of patterns, reduces the possibility of human description errors, and simplifies comprehension of the results. The abstraction makes for easy communications and accurate understanding of overall system structures at a higher level.

2.2.3 Software Architecture Design in the Development Process

The vocabulary gap between requirements and programming is substantial and requires advanced models and notations for the intermediate step; in which case, software architecture emerges to fill this gap. [72]

In the waterfall model, the software architectural level fits between requirements and design. Software architecture focuses on high-level design, whereas the original design stage in the waterfall model focuses on translating requirements and architecture into low-level design [23].

In the software development life cycle, software architecture can be used as a basis for design to satisfy all functional requirements. It facilitates the early detection of reuse opportunities at the architecting stage. In addition, software architecture can serve as a framework for accommodating the change of life-cycle requirements. It can be used as a basis for software maintenance to prevent architectural erosion and architectural drift. "Architectural erosion is due to violations of the architecture", and "architectural drift is due to insensitivity about the architecture" [61].

2.2.4 Architectural Styles

An architectural style defines a family of systems in terms of a pattern of structural organization, and determines the configuration of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [27]. "An architectural style encapsulates important decisions about the architectural elements and emphasizes important constraints on the elements and their relationships." [61]

Typically, an architectural style abstracts elements and formal aspects from various specific architectures, so it is less constrained and less complete than a specific architecture. A number of common architectural styles [24,70,81] have been identified based on aforementioned *pattern recognition* and *abstraction processes* and new styles are still emerging [53,54,77]. Nevertheless, different architectural styles may also be combined into a single design as a heterogeneous architectural style or a hierarchical style with each layer designed in different style. Using architectural styles as architectural representations can promote design reuse, lead to significant code reuse, improve the realization of complicated software systems, and permit specialized style-specific analysis, etc. [24,27]

The following shows a number of identified and commonly used architectural styles [27,81]:

- **Data Flow Style**

  - Batch Sequential.

  - Pipes and Filters. [4,5,68,72] (e.g., Unix pipes, signal processing.)

- **Call and Return Style**

  - Main Program and Subroutines (Explicit Invocation)

  - Hierarchical layers (e.g., OS kernels, ISO OSI.)

  - Data Abstraction and Object-Oriented Organization

- Client-Server (e.g., File servers, distributed databases)

- **Independent Components**

  - Communicating processes (e.g. Many distributed systems.)

  - Event-based, Implicit Invocation (e.g. tools such as editors and variable monitors register for a debugger's breakpoint events.)

- **Virtual Machines**

  - Rule-based systems (e.g., Expert systems.)

  - Table Driven Interpreters (e.g., Blackboard shell, rule-based systems)

- **Data-Centered Systems**

  - Transactional Database Systems

  - Repositories (e.g., Blackboard Systems)

- **Other Architectures:**

  - Parallel Computing Architectures

  - Fault Tolerant/Backup Architectures

  - Heterogeneous Architectures [2,69,72]

  - Domain-Specific Software Architectures [55,79]

2.2.5 Formal Specification Models

Formal specification models are used to solve the problems of many informal approaches. The main objective of these models is to support abstraction idioms or architectural styles commonly used by designers, and to specify packaging properties and functional properties of components. In addition, these models provide an explicit and localized connector including the rules for component interactions, defining an abstraction function to map from code or lower-level constructs to higher-level constructs in respect to externally developed construction and analysis tools [72].

Formal specification models can be served as reusable frameworks if they are extensible and adaptable. The models are extensible when they are easy to use to modify the specification so as to handle new variations in the functions. If the models are relatively easy to use to adapt the specification to different applications, they are adaptable. [24]

The development of formal specification models [3,37,68] involves four steps. The first is informal description, which simply describes the purpose of elements and general domain of utility. The second formalizes abstract syntax for architectures as the basis for specification. The third, for any given style, defines the semantic model to capture the meanings of primitive elements and the rules for composition. The last step demonstrates analysis within and between formally defined architectural styles.

2.2.6 Architectural Description Languages (*ADLs*)

Software architecture is concerned with high-level design issues, and architectural description languages lay the formal basis for this architecting stage of the life cycle. These higher-level languages appear to compensate for the weaknesses of existing programming languages on structure descriptions, and to try to solve the essential problems of software engineering, such as *complexity*, *conformity*, *changeability*, and *invisibility* [12].

Architectural description languages are expected to directly support reuse or *megaprogramming*, rapid prototyping (e.g. the *Rapide* prototyping language and toolset [55]), reengineering, early consideration of system non-functional properties, and comprehension of the overall system. Megaprogramming is the practice of building and evolving computer software component by component. [11]

An ideal architectural description language [1,71] is supposed to provide the following kinds of properties: composition, abstraction, reusability, configuration, heterogeneity, and analysis.

- **Composition:** It should be possible to describe a system as a composition of independent components and connections.
- **Abstraction:** It should be possible to describe the components and their interactions of a software architecture in a way that clearly and explicitly prescribes their abstract roles in a system.
- **Reusability:** It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system.
- **Configuration:** Architectural descriptions should localize the description of system structure, independent of the elements being structured. They should also support dynamic reconfiguration.
- **Heterogeneity:** It should be possible to combine multiple, heterogeneous architectural descriptions.

- **Analysis:** It should be possible to perform rich and varied analyses of architectural descriptions.

2.2.6.1 Existing ADLs

Due to the existing languages' notations, which fail to provide architectural design and description, several new languages with improved notations have been proposed. They were designed for different kinds of scopes of use. Some of them are general purpose, others are dedicated to architectural or system structure description, and the remaining groups are developed for specific domain problems, as shown in Table I. [1]

Scope	Examples
General Purpose	Z, Prolog
Architecture/System Description	Rapide, DICAM, Unicon, Wright, UNAS, Aesop
Domain-Specific	MetaH, ControlH

Table I: Different scope of use among ADLs

- **DICAM** [79], designed by Teknowledge Federal Systems, inc., describes a system's architecture using a collection of schemas. It starts with the built-in types as a foundation, supplies technology for rapid development of high-performance intelligent controllers, and supports tools for graphical and textual entry of architecture.

- **Wright** [6,8], developed at CMU by Allen and Garlen, describes general system architectures, concerned with interactions between components. It detects connector-component type mismatches, and uses Communicating Sequential Processes (CSP) [7,33] as port and role protocols.

- **ControlH** [10,20], designed by Honeywell, is specialized for real-time guidance, navigation and control software, allows modeling and analysis of continuous, time-varying signals, and supports tools including *Ada* generator.

- **MetaH** [10,83], designed by Honeywell, is specialized for real-time multiprocessor embedded software, concerned with scheduling and reliability, and dedicated to systems constructed out of pre-declared classes. It also has heavy tool support as ControlH.

- **Rapide** [52,83], proposed at Stanford University by Luckham et al., describes general system architectures, provides a module description language, uses object-oriented or event-based components for system designs, and supports rapid prototyping through simulation of the architecture.

- **Unicon** [72], developed at CMU by Shaw et al., describes software architectures in general, includes rich set of built-in components and connectors, has interfaces for both components and connectors to be associated with implementations, and supports external analysis tools.

- **UNAS** [63], designed by TRW Space & Defense, includes very simple constructs such as sockets, tasks and circuits but no formal *ADL*, has been modeled in *Z* language, and supports rapid prototyping of architectures.

- **Aesop** [26] describes software architectures in general, supports the definition of new architectural styles as a system of object types, and provides an operational basis for style definition.

- Some other ADLs include **Z** [74,75], **LILEAnna** [80,83], and **QAD** [83], etc.

2.2.7 Domain-Specific Software Architectures (*DSSAs*)

"A domain-specific software architecture is a process and infrastructure that supports the development of a *domain model*, *reference requirements*, and *reference architecture* for a family of applications within a particular problem *domain*. The expressed goal of a *DSSA* is to support the generation of applications within a particular domain." [81]

A *domain model* implies the "terminology and semantics characterizing elements and relationships in a domain." [76] "A *domain* is defined by a set of common problems or functions that applications in that domain can solve or do. Also, a domain is typically characterized by a common jargon or ontology for describing problems or issues that applications in it address." [81] Furthermore, the domain model defines the terms used to express requirements and evaluate systems. The purpose of a domain model is to provide individuals with a means of developing or maintaining applications in a domain so as to understand the various aspects of the domain. [81]

"*Reference architecture* is a (generic) software architecture for a family of application systems." [82] It consists of a reference architecture model, configuration decision tree, architecture schema or design record, reference architecture dependency diagram (topology), components interface descriptions, constraints, and rationale.

"*Reference requirements* are the (generic) behavioral requirements for applications in a domain." [82] Besides specifying the functional requirements identified in the domain model, it also contains non-functional, design, and implementation requirements.

2.2.7.1 Motivation and Approaches

The primary motivation of software development using *DSSAs* is to reduce time and cost of producing specific application systems within a supported domain as well as increase product quality. In addition, *DSSAs* improve manageability and productivity, position for acquisition of future business, handle complexity as well as reliability, and replace custom fabrication by construction from components (i.e. reuse previous solutions then swap into newer version of components). [15,76]

Furthermore, in order to meet the goals of *DSSAs*, software development relies on the identification and deep understanding of a selected domain of applications. It also requires a variety of support tools, including repository mechanisms, prototyping facilities, and analysis tools, etc. [76] Accordingly, some important tools and their functions are pointed out by Clark [15] as shown in Table II. Similarly, some other less important support tools may also be required such as requirement verification, language processing and compiling, user interfaces, databases, distributed and real-time operating systems, configuration management, and documentation tools.

Another key approach to meet the goals of *DSSAs* is through *software reuse.* Software reuse is the use of existing software artifacts when building new software systems in order to reduce the time and effort required, while also improving the quality. It is possible to meet the goal through software reuse based on parameterization of generic components and interconnection of components within a canonical solution *framework.* A framework is provided by reference architecture in which we can configure diverse software components and coordinate their activities.

Tool Type	Functional Capability
Modeling	- Identify domain terminology - Describe the behavior of objects, attributes and relationships in domain model - Explicitly define the boundaries of the domain - Provide some simulation or inferencing capability
Requirements Management	- Describe new application's objects, attributes and relationships - Make evaluation criteria and constraints explicit - Capture rationale
Architecture Specification	- Define reference architecture - Prescribe (describe, constrain) components and connectors, behaviors - Explicitly show platform capabilities
Architecture Specialization and Application Evolution	- Assist in iteration of application design and development process - Configure reference architecture to meet application's needs - Specify and instantiate components - Compose candidate configurations into an executable form - Assess and evaluate candidate configurations - Capture rationale
Repository	- Store and retrieve components
Component Selection	- Assist in choosing from several alternatives
Component Generators	- Transform specifications into executable code
Configuration, Load and Exercise	- Assist in the integration and configuration of components

Table II: Tool support for DSSA generation

Chapter 3

Model Foundations

The theory of Markov chains has been applied in modeling hardware and software behaviors, and several Markov-based approaches [14,44,48] were developed for measuring software reliability. The application of a Markov usage model can be seen in the construction of *IBM's DB2* [84] and in a number of tools listed in the survey of software tools for evaluating reliability, availability, and serviceability [39].

3.1 Architecture-Based State Model

Our objectives are to model the reliability of large-scale software, support design decision-making, and facilitate quality prediction at the early design stage. We found Software Reliability Growth Models (SRGMs) to be unsuitable because of the needs of test data, available only at the late stage of software life cycle. The alternative is to take advantage of the white-box approach of discrete time Markov-based reliability models. Although the models can support decision-making and facilitate early prediction, they are insufficient to model complex structures and address execution history.

Therefore, we developed an architecture-based state model, allowing us to take into account several issues at an early phase of the software process: *complex system structures, heterogeneous architectures, execution history, and software behaviors both probabilistic and deterministic.* Our architecture-based state model is based on discrete-time Markov models, whose properties are described in Appendix A. Our state model, taking into account architectural styles and execution history, resolves the modeling limitations of homogeneous Markov-based reliability models [14], discussed in Appendix B. In Appendix C, one sample example of utilizing traditional approaches to compute system reliability is depicted.

The fundamental difference between our state model and the traditional discrete time Markov-based reliability models is that our state model allows multiple components to aggregate into a state or a component to be executed in multiple states. Such a definition facilitates the modeling of heterogeneous system structures, both probabilistic and deterministic software behaviors, and execution history. Chapter 4 illustrates the utilization of architectural styles to realize system structures, and Chapter 5 models software behaviors and execution history. Basically, software is first studied to realize its system structures through the identified architectural styles. The characteristics and behaviors of these styles are then addressed into our state model and system reliability can be calculated accordingly.

Given that a *condition* is an instance of a set of components waiting for execution and a *circumstance* is an event that can activate the execution of a component, we define our state model as below:

- A *state* is a set of *circumstances* characterizing a system at a given *condition*.

- A *transition* is a passage from one state to another and the *transition probability* is the likelihood of the *transition* being triggered.

Definition 3.3.1: Given an event is an occurrence of activation to a component, we define a state $s_i = (E^i, C^i)$.

C^i: a set of m components $\{c_1^i, c_2^i, \ldots, c_m^i\}$, and

E^i: a set of m events $\{e_j^i \mid e_j^i$ is to activate $c_j^i\}$, for $1 \leq j \leq m$.

Definition 3.3.2: We denote our state model S_M by a 5-tuple (Q, δ, s_1, s_k, M).

Q: a finite set of k states $\{s_1, s_2, \ldots, s_k\}$.

δ: a transition function mapping $Q \times E$ to Q.

$$E = \{E^i \mid E^i \text{ in } s_i,\ 1 \leq i \leq k\}, \text{ where } s_i = (E^i, C^i).$$

s_1: the *initial* state to start the transition.

s_k: the *final* state that terminates the transition to any states in Q.

M: a $k \times k$ transition matrix, constructed from those states in Q. Each entry $M(i, j)$ is the transition probability from state s_i to state s_j.

- $M(i, j) \neq 0$, if $\delta(s_i, E^i) = s_j$, $1 \leq i < k$, and $1 \leq j \leq k$.
- $M(i, j) = 0$, otherwise.

The major step of our architecture-based reliability model is to build the transition matrix M, taking into account system structures and software behaviors. With this transition matrix M, when a standardized stochastic matrix can be constructed, discussed in Appendix A, then $(I\text{-}M)^{-1}$ is nonsingular. Therefore, we can compute software reliability following the traditional approaches as $R = (-1)^{k+1} \frac{|(I-M)_{k,1}|}{|I-M|} R_k$, the formula {B2} discussed in Appendix B. I is an identity matrix, $|I\text{-}M|$ is the determinant of matrix $(I\text{-}M)$, and $|(I\text{-}M)_{k,1}|$ is the determinant of the minor matrix, excluding the last row and the first column of the matrix $(I\text{-}M)$. In traditional models, R_k is the reliability of the component executed in state s_k. However, in our model R_k represents the overall reliability of the component(s) executed in state s_k, since multiple components can be executed in the same state.

The following shows the construction of a standardized stochastic matrix T, with each row sum equal to 1. Similar to the traditional Markov-based reliability models in Appendix B, two absorbing states S and F are added to represent a successful state and a failure state, respectively. The transition matrix M, with each row sum less or equal to 1, is a $k \times k$ sub-matrix in the stochastic matrix T. All of the states in M are transient states. Each state will eventually transit to either the successful state S, or the failure state F after a certain number of direct or indirect transitions. When the transition is eventually to the successful state S, the software succeeds. Otherwise, the software fails when the transition is to the failure state F.

$$T = \begin{array}{c|ccc} & S & F & s_1 \cdots s_k \\ \hline S & 1 & 0 & 0 \\ F & 0 & 1 & 0 \\ s_1 \cdots s_k & B_1 & B_2 & M \end{array}, \quad B_1 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ R_k \end{bmatrix}, \quad B_2 = \begin{bmatrix} 1 - \sum_{j=1}^{k} M(1,j) \\ \vdots \\ 1 - \sum_{j=1}^{k} M(k-1,j) \\ 1 - R_k \end{bmatrix}$$

The matrices B_1 and B_2 are $k \times 1$ matrices. In the transition matrix M, only the final state s_k can directly transit to the successful output state S. Therefore, all the entries of B_1 are equal to 0, except the last row entry $B_1(k,1)$ is equal to R_k, the overall reliability of the component(s) in state s_k. Based on the definition of our state model S_M, the last row sum of M is equal to 0. Therefore, $B_2(k,1)$ is equal to $1-B_1(k,1)$ and each entry $B_2(i,1)$, $1 \leq i < k$, is equal to 1 minus the ith row sum of M.

In the modeling, if a component executed in the final state has a chance to invoke other components in the other states, this original final state cannot ensure the termination of the state transitions in M. In this situation, we construct a new final state to the M matrix that only transits to states S and F, and link the original final state to the new final state, based on the probability. This fulfills that the final state terminates the execution of software, and all states in M are transient. In this new final state, we have a virtual component always executed reliably so that there is no reliability difference to the original model.

3.2 The Modeling of Multiple Initial States and Multiple Final States

We have introduced the reliability modeling of software with one initial state and one final state. Typically, a system can have multiple initial states and/or multiple final states instead of only one initial state and only one final state. In this section, we present a simple scenario to refine the problem of multiple initial states and multiple final states to one single initial state and one single final state by introducing a *super-initial* state S^I and a *super-final* state S^F to the state diagram.

Suppose a system has a set of initial states $I = \{S_1^i, S_2^i, \ldots\ldots, S_p^i\}$ and a set of final states $F = \{S_1^f, S_2^f, \ldots\ldots, S_q^f\}$. We add a directed edge (S^I, S_j^i) with transition probability P_{Ij}, observed from the operational profile, for each $j = 1, 2, \ldots., p$. Similarly, we add a directed edge (S_j^f, S^F) with a transition probability P_{jF} for each $j = 1, 2, \ldots., q$. Since S^I and S^F are two added states, we assume that the components in these states are virtual components with component reliabilities both assigned with 1 without affecting the measurement of system reliability. Note that the final state S^F only transits to either the successful output state S, or the failure state F in the stochastic matrix T to ensure that all states in M are transient; i.e. the row sum of M is equal to 0. Figure 1 shows how a state diagram with multiple initial states and final states in the dotted rectangular area can be converted to a state diagram with only a single initial state and a single final state.

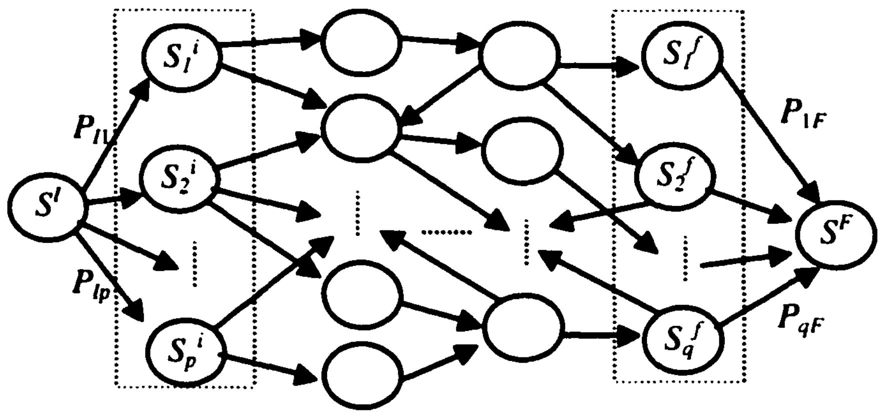


Figure 1: One super-initial state S^I and one super-final state S^F

Chapter 4

Heterogeneous Software Reliability Modeling and Component Sensitivity Measurement

One of the objectives of our architecture-based software reliability model is to model the reliability of large-scale software, taking into account the characteristics of complexity of architectures. The architectures are commonly composed of a variety of components and interfaces, which can form different configurations. Each configuration can have different topologies and constraints. Therefore, the system structures can be heterogeneous. The major concerns at the design phase of large-scale software are commonly the overall system structures. Because the change of architectural designs at a later phase of the software life cycle can be difficult and expensive, it is thus important to make a good design decision in the early design phase, and software architecture serves as an important attribute in our reliability model.

Software architecture is commonly accepted as the composition of components, connectors, and configurations [27]. *Components* define the locus of computation. *Connectors* define the interactions (e.g. procedure call, data flow, implicit triggering, message passing, shared data, and instantiation [71]) between components. *Configurations* define the topology and configuration constraints of the components and connectors. The theme of our study is to evaluate the reliability of software based on its architecture. Software architectural styles can characterize software systems that share certain common properties, such as the structure of organizations, constraints, and high-level semantics [25]. Therefore, architectural styles allow us to comprehend complex system structures in a formal way for both homogeneous and heterogeneous architectures.

An architectural style is a high level abstraction of repeatable design patterns that designers can use to describe software systems and communicate with programmers. A number of architectural styles have been identified and used to facilitate communications and to improve the understanding of software systems [27,81]. For example, *client-server* architecture means that the architecture of a software is based on the client-server style. This idiom "*client-server*" implies that a system has client components and server components. The connector between a client and a server is through a network. The configurations regulate that a client can ask multiple servers for services, a server can serve multiple clients, and a client can also be a server. Therefore, such a high-level abstract term can simplify the detailed descriptions and facilitate communications among designers and programmers.

4.1 Reliability Modeling

Three attributes are required in our software reliability modeling. One is the system architecture, another is the set of component reliabilities, and the other is the set of transition probabilities for every two connecting components. The system architecture is classified into architectural styles, in order to realize the different interactions and intercommunications. The reliability of a component can be measured by traditional approaches [45,49,58] or the inter-component dependency approach proposed by Hamlet et al [32]. The transition probabilities are observed from the operational profile and are independent of component reliabilities.

Here we show the modeling steps of our architecture-based software reliability model. For a given software, we:

1. Identify the architectural styles in a system to understand the characteristics and complexity of the system structures,
2. Construct a state model, encompassing all the identified styles, and
3. Use the state model to compute software reliability.

For the first modeling step, please reference [24,27,81] for the identification of architectural styles in a system. In our reliability model, we utilize the identified architectural styles to recognize the system structures, transform the architecture into a state model, and then compute the system reliability accordingly. From Section 4.1.1 to 4.1.4, we use four architectural styles, batch-sequential, parallel/pipe-filter, call-and-return, and fault tolerance styles, as examples, to demonstrate how to construct the state model for each style. These four styles are commonly used with varying characteristics, and other styles can be extended with certain modifications. In Section 4.1.5, we construct a state model for heterogeneous software with the combination of these discussed styles.

From our state model in Section 3.1, a *state* is a set of *circumstances* characterizing a system at a given *condition*, and the *transition probability* is the probability of going through a *transition*. In our state model, R_i represents the reliability of component c_i, L_{ij} represents the reliability of connector from component c_i to component c_j, and P_{ij} represents the transition probability from component c_i to its successor component c_j.

4.1.1 Batch-Sequential Style

In the batch-sequential style, components are executed in a sequential order and outputs of a component are produced only after all its inputs are fully processed. In other words, only a single component is executed in any instance of time. Upon the completion of the execution, the control authority transfers from the executed component to one (and only one) of its successors. The selection of the succeeding component can be probabilistic (if more than one successors) or deterministic (if only one successor).

The state model of batch sequential style software can be modeled as follows: Since the execution of the current component must be fully completed before the next component can proceed, only one component is executed at a time. The transformation from the architecture to a state model can be viewed as a mapping of a single component to a state, which represents an execution of the component. A transition from one state to another takes place after a component is activated by an event, and the control authority is also transferred.

An instance of this style can be modeled as shown in Figure 2(a), where c_1, c_2, ... , c_k are software components and a branching component, such as c_2, can only transfer control to one of its subsequent components. With $s_i = (E^i, C^i)$ and state model S_M defined in Section 3.1, this batch-sequential style software has states $s_1 = (\{e_1^1\}, \{c_1\})$, $s_2 = (\{e_1^2\}, \{c_2\})$, ..., and $s_k = (\{e_1^k\}, \{c_k\})$, and state transitions $\delta(s_1, \{e_1^2\}) = s_2$, $\delta(s_2, \{e_1^3\}) = s_3$, $\delta(s_2, \{e_1^4\}) = s_4$, ..., and $\delta(s_{k-1}, \{e_1^k\}) = s_k$. When the transition reaches another state s_i, event e_1^i is to activate component c_i, $1 \le i \le k$. We construct the state model $S_M = (Q, \delta, s_1, s_k, M)$, where $Q = \{s_1, s_2, ..., s_k\}$. Figure 2(b) is the state diagram of this instance of the batch-sequential style, where s_1, s_2, ... , s_k are the mapping states to components c_1, c_2, ... , c_k.

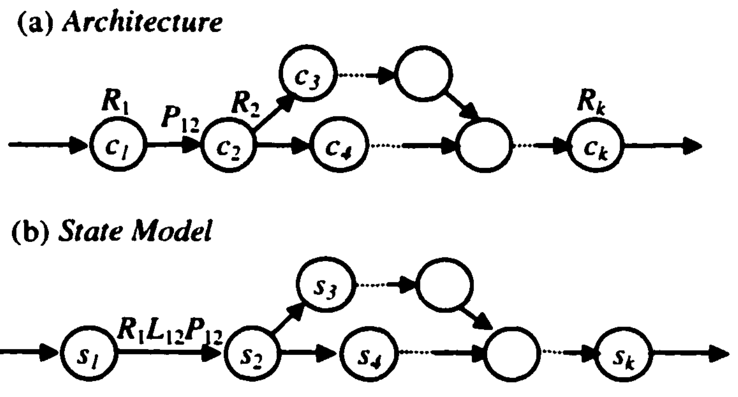


Figure 2: Batch-sequential style

From the state diagram in Figure 2(b), a $k \times k$ matrix M can be constructed as below, which is similar to traditional homogeneous Markov-based reliability models in Appendix B. The transition probability from one state s_i to another state s_j is equal to

$R_iL_{ij}P_{ij}$, which is the product of the component reliability R_i of component c_i in state s_i, the connector reliability L_{ij} from component c_i to c_j, and the transition probability P_{ij} from component c_i to c_j.

$$\begin{cases} M(i,j) = R_i L_{ij} P_{ij},\ \delta(s_i, E^j) = s_j \\ M(i,j) = 0, \text{ otherwise} \end{cases}, \text{ for } 1 \le i < k,\ 1 \le j \le k \quad \ldots\ldots \{1\}$$

4.1.2 Parallel/Pipe-Filter Style

In a concurrent execution environment, components commonly run simultaneously to improve system performance. The parallel or pipe-filter styles are used to model this type of systems. In these styles, a set of components cooperates and works concurrently to fulfill a task. Each component works on a partition or a subtask. The main difference between these two styles is that parallel computation is generally in a multi-processor environment, whereas pipe-filter style occurs commonly in a single processor, multi-process environment.

The state model of parallel/pipe-filter style software can be modeled as follows: If only one single component is running at a given condition, a state represents an execution of the component. Otherwise, we model the scenario of the set of concurrent components by a state spanning the time from the beginning to the completion of the executions of these concurrent components. A transition from one state to another takes place after a set of components is activated, and the control authority is transferred. The transformation from the architecture to the state model can be viewed as a mapping of a single component to a state, or multiple concurrent components to a state.

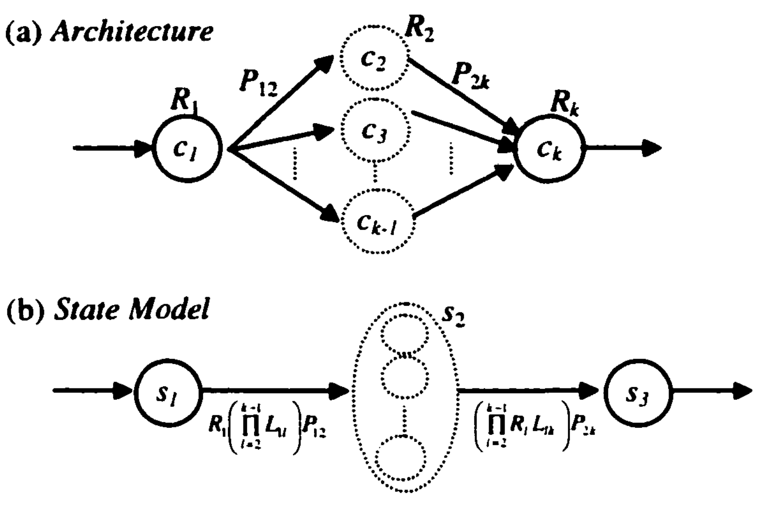


Figure 3: Parallel or pipe-filter style

An instance of this parallel/pipe-filter style can be modeled as shown in Figure 3(a), in which the dotted circles are components c_2 to c_{k-1} running in parallel. Each parallel component works on a portion of the outcome coming from component c_1 and then transfers the execution to the next common subsequent component c_k in synchronization with other parallel components. With $s_i = (E^i, C^i)$ and state model S_M defined in Section 3.1, this parallel style software has three states $s_1 = (\{e_1^1\}, \{c_1\})$, $s_2 = (\{e_1^2, e_2^2, ..., e_{k-2}^2\}, \{c_2, c_3, ..., c_{k-1}\})$, and $s_3 = (\{e_1^3\}, \{c_k\})$, and two possible state transitions $\delta(s_1, \{e_1^2, e_2^2, ..., e_{k-2}^2\}) = s_2$, and $\delta(s_2, \{e_1^3\}) = s_3$. When the transition reaches states s_1 or s_3, events e_1^1 and e_1^3 are to activate components c_1, and c_k, respectively. When the transition reaches state s_2, events $e_1^2, e_2^2, ..., e_{k-2}^2$ are to activate components $c_2, c_3, ..., c_{k-1}$. We construct $S_M = (Q, \delta, s_1, s_3, M)$, where $Q = \{s_1, s_2, s_3\}$. Figure 3(b) is the state diagram of this instance of the parallel/pipe-filter architectural style. The execution of component c_1 is one single component running, so is component c_k. They are mapped to their individual states s_1 and s_3. The executions of components c_2

to c_{k-1} are congregated into state s_2 representing multiple components running concurrently.

From the state diagram in Figure 3(b), a 3×3 matrix M can be constructed as below. Based on the characteristics of parallel/pipe-filter style, the transition probabilities from component c_1 to components c_2, c_3, ..., and c_{k-1}, are all equal to P_{12}. Likewise, the transition probabilities from components c_2, c_3, ..., and c_{k-1} to component c_k are all equal to P_{2k}. The entry $M(1,2)$ is the transition probability from state s_1 to s_2 equal to $R_1\left(\prod_{l=2}^{k-1} L_{1l}\right)P_{12}$, in which $\left(\prod_{l=2}^{k-1} L_{1l}\right)$ means that the connectors from component c_1 to all the components c_2, c_3,...., and c_{k-1} need to be reliable. Entry $M(2,3)$ means that all the components from c_2 to c_{k-1} in state s_2 perform successfully and finally reach state s_3. Therefore, the value of $M(2,3)$ is equal to $\left(\prod_{l=2}^{k-1} R_l L_{lk}\right)P_{2k}$, which is the product of all the components operating reliably in this state, the connector reliabilities from components c_2, c_3, ..., and c_{k-1} to component c_k, and the transition probability from components c_2 to c_k.

$$\begin{cases} M(1,2) = R_1\left(\prod_{l=2}^{k-1} L_{1l}\right)P_{12}, \ \delta(s_1,\{e_1^2,e_2^2,\ldots,e_{k-2}^2\}) = s_2 \\ M(2,3) = \left(\prod_{l=2}^{k-1} R_l L_{lk}\right)P_{2k}, \delta(s_2,\{e_1^3\}) = s_3 & \text{, for } 1 \leq i < 3,\ 1 \leq j \leq 3 \ \ldots\ldots\ldots\ldots\ \{2\} \\ M(i,j) = 0, \text{ otherwise} \end{cases}$$

4.1.3 Fault Tolerant Style

Fault-tolerant architectural style software consists of a set of components compensating for the failure of the primary component. When the primary component fails, the first backup component will take over the responsibility and become the new primary component. If this new one fails as well, another backup component will take over. The implementation of these fault-tolerant components may involve using different algorithms and data structures to improve the system reliability. Therefore, the reliability of components in the same fault-tolerant set can be different from each other. In this style, components that compensate for the failure of each other are aggregated into a state.

The state model for fault tolerant style software can be modeled as follows: If only one single component is running, a state represents an execution of a component. Otherwise, we model the scenario of fault tolerant components by a state spanning the time from the beginning of the primary component to the completion of the activated fault tolerant components. A transition from one state to another takes place after finishing and stopping the activation of a set of components, and the control authority is transferred. The transformation from the architecture to a state model can be viewed as a mapping of a single component to a state, or multiple fault tolerant components to a state.

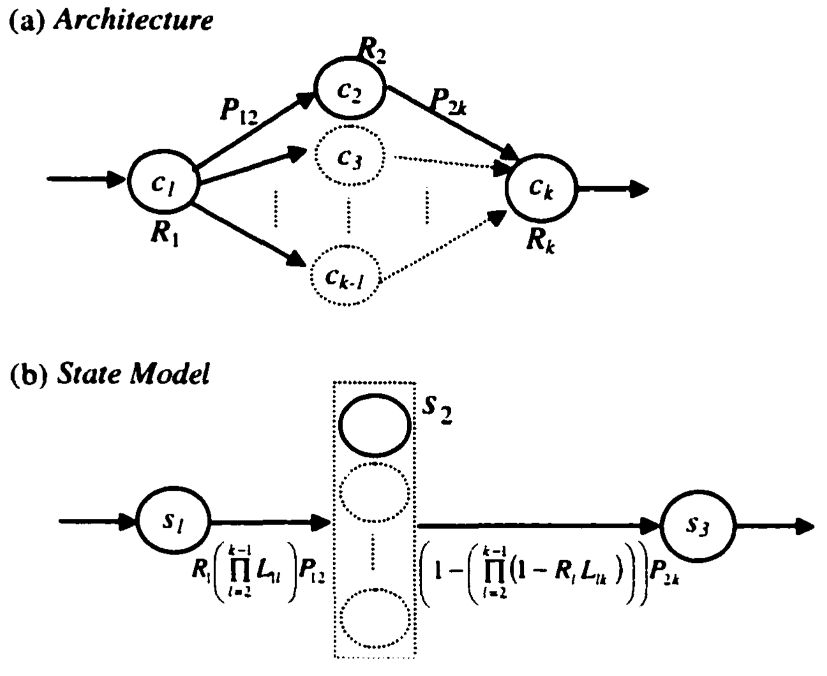

(a) *Architecture*
P_{12}
R_2
c_2
P_{2k}
c_1
c_3
c_k
R_1
R_k
c_{k-1}
(b) *State Model*
s_2
s_1
s_3
$R_1\left(\prod_{l=2}^{k-1} L_{1l}\right)P_{12}$
$\left(1-\left(\prod_{l=2}^{k-1}(1-R_l L_{lk})\right)\right)P_{2k}$

Figure 4: Fault tolerance style

Here, we discuss one type of scenario where software fails only when all the fault tolerant components fail. Given an instance of this architectural style as displayed in Figure 4(a), those dotted components from c_3 to c_{k-1} are modeled as backup components to the primary component c_2. Inside the dotted rectangle shown in Figure 4(b), these components are congregated into the state s_2 to represent multiple components running as fault tolerance. With $s_i = (E^i, C^i)$ and state model S_M defined in Section 3.1, this fault tolerant style software consists of three states $s_1 = (\{e_1^1\}, \{c_1\})$, $s_2 = (\{e_1^2, e_2^2, \ldots, e_{k-2}^2\}, \{c_2, c_3, \ldots, c_{k-1}\})$, and $s_3 = (\{e_1^3\}, \{c_k\})$, and two possible state transitions $\delta(s_1, \{e_1^2, e_2^2, \ldots, e_{k-2}^2\}) = s_2$, and $\delta(s_2, \{e_1^3\}) = s_3$. When the transition reaches state s_1 or s_3, events e_1^1 and e_1^3 are to activate components c_1, and c_k, respectively. When the transition reaches state s_2, events e_1^2 is to activate component c_2. If component c_2 fails, event e_2^3 is then to activate component c_3. These phenomena will continue until the components in state s_2

all fail. We construct $S_M = (Q, \delta, s_1, s_3, M)$, where $Q = \{s_1, s_2, s_3\}$. Figure 4(b) is the state diagram for this instance of the fault tolerant style. The execution of component c_1 is one single component running, so is component c_k. They are mapped to their individual states s_1 and s_3. The executions of components c_2 to c_{k-1} are congregated into state s_2 representing multiple components running as fault tolerance.

From the state diagram in Figure 4(b), a 3×3 matrix M can be constructed as below. The characteristics of the fault tolerant style are similar to the parallel style, where the transition probabilities from component c_1 to components c_2, c_3, ..., and c_{k-1}, are all equal to P_{12}. Likewise, the transition probabilities from components c_2, c_3, ..., and c_{k-1} to c_k are all equal to P_{2k}. For a set of fault tolerant components, they first receive control and then wait for activation. Therefore, the connectors to the set of fault tolerant components from state s_1 to state s_2 are executed. Thus, $M(1,2)$ is equal to $R_l\left(\prod_{l=2}^{k-1} L_{1l}\right)P_{l2}$. The value of $M(2,3)$ is equal to $\left(1-\left(\prod_{l=2}^{k-1}(1-R_l L_{lk})\right)\right)P_{2k}$. Here, $\left(1-\left(\prod_{l=2}^{k-1}(1-R_l L_{lk})\right)\right)$ is the overall reliability of the set of fault tolerant components and their subsequent matching connectors, based on the removal of the possibility that not a single fault tolerant component and its subsequent matching connector succeed.

$$\begin{cases} M(1,2) = R_1\left(\prod_{l=2}^{k-1} L_{1l}\right)P_{12}, \delta(s_1,\{e_1^2, e_2^2, \ldots, e_{k-2}^2\}) = s_2 \\ M(2,3) = \left(1-\left(\prod_{l=2}^{k-1}(1-R_l L_{lk})\right)\right)P_{2k}, \delta(s_2,\{e_1^3\}) = s_3 \text{, for } 1 \le i < 3,\ 1 \le j \le 3 \quad \ldots\ldots\ldots\ldots\ldots \{3\} \\ M(i,j) = 0, \text{ otherwise} \end{cases}$$

4.1.4 Call-and-Return Style

In a call-and-return style, a caller component can request services provided by the callee components. Before the services are fulfilled by the callee components, the control remains on the caller component. After that, the caller component resumes the execution from where it left, and transits to the next subsequent component. Therefore, the callee components may be executed many times with only one time execution of the caller component.

The state model for call-and-return style software can be modeled as follows: A state represents an execution of a component. A transition from one state to another takes place after a component is activated by an event and the control authority is also transferred, or when the execution temporarily transfers to its callee component. The transformation from the architecture to the state model can be viewed as a mapping of a component to a state.

An instance of this call-and-return style can be modeled as shown in Figure 5(a). Component c_1 is a caller component, and its callee is component c_2. A one time execution of c_1 can invoke c_2 many times before component c_1 finally transits to component c_3. With $s_i = (E^i, C^i)$ and state model S_M defined in Section 3.1, this call-and-return style software has three states $s_1 = (\{e_1^{\,1}\}, \{c_1\})$, $s_2 = (\{e_1^{\,2}\}, \{c_2\})$, and $s_3 = (\{e_1^{\,3}\}, \{c_3\})$, and three possible state transitions $\delta(s_1, \{e_1^{\,2}\}) = s_2$, $\delta(s_2, \{e_1^{\,1}\}) = s_1$, and $\delta(s_1, \{e_1^{\,3}\}) = s_3$. This represents that c_1 can call c_2 many times before c_1 transits to c_3. When the transition reaches state s_i, event $e_1^{\,i}$ is to activate component c_i, $1 \le i \le 3$. From the above derived states and possible transitions, we have $S_M = (Q, \delta, s_1, s_3, M)$, where $Q = \{s_1, s_2, s_3\}$.

Figure 5(b) is the state diagram of this instance of the call-and-return architectural style. The execution of component c_1 is one single component running at a given condition, so is component c_2 and c_3. Therefore, they are one-to-one mapped to their individual states s_1, s_2 and s_3.

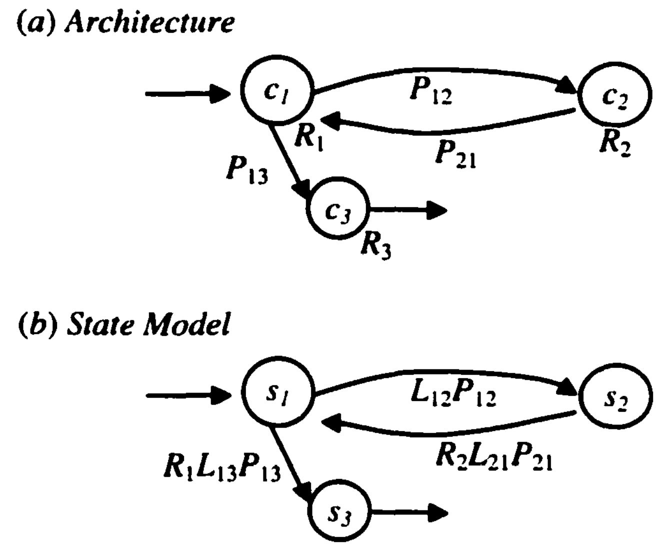


Figure 5: Call-and-return style

For the state diagram in Figure 5(b), a 3 × 3 matrix M can be constructed as below. The entry $M(1,3)$ is equal to $R_1L_{13}P_{13}$, which is the product of the reliability of caller c_1, the connector reliability from c_1 to c_3, and the transition probability from c_1 to c_3. Likewise, the entry $M(2,1)$ can be computed as $R_2L_{21}P_{21}$, which is the product of the reliability of callee c_2, the connector reliability from c_2 to c_1, and the transition probability from c_2 to c_1. The critical entry is $M(1,2)$, which is equal to $L_{12}P_{12}$ instead of $R_1L_{12}P_{12}$. This entry considers only the connector reliability and the transition probability from c_1 to c_2 without considering the reliability of the caller c_1. That is because component c_1 is only executed once before transiting to component c_3, regardless of how many times component c_2 is executed. Therefore, the entry $M(1,2)$ will not consider the reliability of caller c_1, but the component reliability of this one time execution of c_1 will be considered in the entry $M(1,3)$ when state s_1 transits to state s_3.

$$\begin{cases} M(1,2) = L_{12}P_{12},\ \delta(s_1,\{e_1^2\}) = s_2, c_1 \text{ is a caller in } s_1 \\ M(2,1) = R_2L_{21}P_{21},\ \delta(s_2,\{e_1^1\}) = s_1 \\ M(1,3) = R_1L_{13}P_{13},\ \delta(s_1,\{e_1^3\}) = s_3 \\ M(i,j) = 0, \text{ otherwise} \end{cases} \text{, for } 1 \le i < 3,\ 1 \le j \le 3 \ \ldots\ldots\ldots\ldots \{4\}$$

4.1.5 Heterogeneous Architecture Modeling

Our state model takes into account system structures and is applicable to model the combinations of different architectural styles. In this section, we formalize the construction of the transition matrix M for the reliability modeling of heterogeneous software with the aforementioned four types of architectural styles. When a system has more styles involved, this transition matrix can be further extended. Take the styles listed in Section 2.2.4 for example. Client-server architecture and the method invocation of object-oriented software are similar to call-and-return style. However, the connector reliabilities may vary, especially when network transmissions are involved. Besides, the uncertainties in the implicit invocation or event-driven styles can be modeled similar to the branching characteristics in the batch-sequential style, using probabilities to address uncertainties.

Assume that total x components are in software system G. After the architecture-to-state transformation, we obtain a state set ξ which consists of n states transformed from batch-sequential, parallel/pipe-filter, fault-tolerant, or/and client-server styles. Thus, we have

$G = \{c_\tau \mid \text{component } c_\tau \in B \cup P \cup F \cup S \cup C, 1 \le \tau \le x\}$

B: set of components in batch sequential style.

P: set of components in parallel style.

F: set of components in fault-tolerant style.

S: set of callee components in call-and-return style.

C: set of caller components in call-and-return style.

$\xi = \xi_B \cup \xi_P \cup \xi_F \cup \xi_C \cup \xi_S = \{s_1, s_2, \ldots, s_n\}, |\xi| = n.$

, where

$\xi_B = \{s_i \mid c_\tau \text{ uses state } s_i, c_\tau \in B\}$, states for batch sequential components

$\xi_P = \{s_i \mid P_j \text{ uses state } s_i, P_j \subset P, 1 \le j \le u\}$, states for u parallel component sets

$\xi_F = \{s_i \mid F_j \text{ uses state } s_i, F_j \subset F, 1 \le j \le v\}$, states for v fault tolerant component sets

$\xi_C = \{s_i \mid c_\tau \text{ uses state } s_i, c_\tau \in C\}$, states for caller components

$\xi_S = \{s_i \mid c_\tau \text{ uses state } s_i, c_\tau \in S\}$, states for callee components

The transition matrix developed from the state view is described as follows. Let s_i, $s_j \in \xi$. We define the $n \times n$ transition matrix M, which labels all the states on the row and the column, and entry $M(i, j)$ can be computed based on the following criteria:

$M(i, j) =$

$$\begin{cases} 0 & \text{if } \delta(s_i, E^j) = s_i \\ R_i L_{ij} P_{ij} & \text{if } s_i \notin \xi_P \cup \xi_F \text{ and } s_j \notin \xi_S \\ \left(\prod_{k=a+1}^{a+p} R_k L_{kj}\right) P_{(a+1)j} & \text{if } s_i \in \xi_P \text{ and } c_{a+1} \text{ to } c_{a+p} \text{ are in } s_i \\ \left(1 - \left(\prod_{k=b+1}^{b+q} \left(1 - R_k L_{kj}\right)\right)\right) P_{(b+1)j} & \text{if } s_i \in \xi_F \text{ and } c_{b+1} \text{ to } c_{b+q} \text{ are in } s_i \\ R_i \left(\prod_{k=c+1}^{c+r} L_{ik}\right) P_{i(c+1)} & \text{if } s_j \in \xi_P \cup \xi_F \text{ and } c_{c+1} \text{ to } c_{c+r} \text{ are in } s_j \\ L_{ij} P_{ij} & \text{if } s_i \in \xi_C \text{, and } s_j \in \xi_S \end{cases} \text{, for } 1 \le$$

$i < n$, $1 \le j \le n$, where $|\xi| = n$ .... {5}

In our model, the state model is utilized to compute software reliability. As discussed in the model foundation in Chapter III, our stochastic transition matrix T is standardized. Therefore, once the transition matrix M is available, software reliability can be computed as $R = (-1)^{n+1} R_n \frac{|(I - M)_{n,1}|}{|I - M|}$, where n is the total number of states transformed from the original x components.

4.1.6 An Example

Figure 6(a) is the directed graph of an architecture view with fifteen components, where c_1 is the input component and c_{15} is the output component. Components c_3 and c_4 are running in parallel. Component c_{10} is a fault tolerant component of c_9. Components c_{11},

c_8, and c_5 are running as call-and-return style, where c_8 and c_5 are the callee components of caller components c_{11} and c_8, respectively. The others are running in sequential manner.

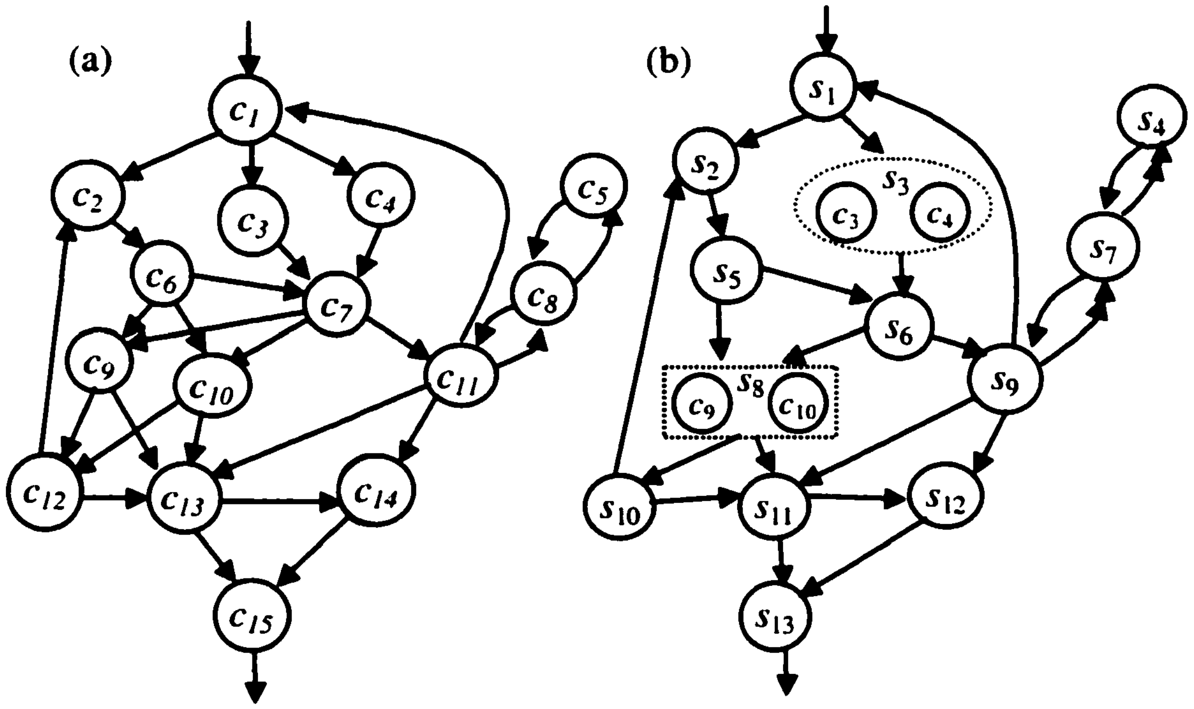


Figure 6: Architecture view with 15 components vs. state diagram with 13 states

The reliability R_i of each component c_i is shown below:

$R_1 = 0.998 \quad R_2 = 0.990 \quad R_3 = 0.980 \quad R_4 = 0.995 \quad R_5 = 0.999$

$R_6 = 0.985 \quad R_7 = 0.996 \quad R_8 = 0.975 \quad R_9 = 0.990 \quad R_{10} = 0.998$

$R_{11} = 0.950 \quad R_{12} = 0.965 \quad R_{13} = 0.970 \quad R_{14} = 0.980 \quad R_{15} = 0.992$

The reliabilities of connector L_{ij} are equal to 1 except the follows:

$L_{6,7} = 0.99 \quad L_{11,8} = 0.99 \quad L_{8,5} = 0.98$

The reaching probabilities P_{ij} between the components c_i and c_j are as follows:

$P_{1,2} = 0.40$	$P_{1,3} = P_{1,4} = 0.60$	$P_{2,6} = P_{3,7} = P_{4,7} = P_{5,8} = 1.0$	
$P_{6,7} = 0.10$	$P_{6,9} = P_{6,10} = 0.90$	$P_{7,11} = 0.25$	$P_{7,9} = P_{7,10} = 0.75$
$P_{8,5} = 0.20$	$P_{8,11} = 0.80$	$P_{9,12} = P_{10,12} = 0.70$	$P_{9,13} = P_{10,13} = 0.30$
$P_{11,1} = 0.15$	$P_{11,8} = 0.20$	$P_{11,13} = 0.50$	$P_{11,14} = 0.15$
$P_{12,2} = P_{12,13} = 0.50$	$P_{13,14} = 0.40$	$P_{13,15} = 0.60$	$P_{14,15} = 1.00$

The transformed state view is shown in Figure 6(b), where a circle represents a state, a dotted oval represents a state for parallel components running concurrently, a dotted rectangle represents a state for fault-tolerant components, an arrow represents the transition from one state to another state, and a double arrow represents the call-and-return style.

With the definitions of $s_i = (E^i, C^i)$ and S_M, this heterogeneous style software has the following thirteen states. In general, a state is enumerated with a number as its index, but component c_i is not definitely mapped to state s_i. Note that state s_3 is a state for a set of two parallel components c_3 and c_4, and state s_8 is a state for a set of two fault tolerant components c_9 and c_{10}.

$s_1 = (\{e_1^1\}, \{c_1\})$	$s_2 = (\{e_1^2\}, \{c_2\})$	$s_3 = (\{e_1^3, e_2^3\}, \{c_3, c_4\})$
$s_4 = (\{e_1^4\}, \{c_5\})$	$s_5 = (\{e_1^5\}, \{c_6\})$	$s_6 = (\{e_1^6\}, \{c_7\})$
$s_7 = (\{e_1^7\}, \{c_8\})$	$s_8 = (\{e_1^8, e_2^8\}, \{c_9, c_{10}\})$	$s_9 = (\{e_1^9\}, \{c_{11}\})$
$s_{10} = (\{e_1^{10}\}, \{c_{12}\})$	$s_{11} = (\{e_1^{11}\}, \{c_{13}\})$	$s_{12} = (\{e_1^{12}\}, \{c_{14}\})$

$s_{13} = (\{e_1^{13}\}, \{c_{15}\})$		

From the above states, the following shows 22 possible transitions between two states out of those 13 states:

$\delta(s_1, \{e_1^2\}) = s_2$	$\delta(s_1, \{e_1^3, e_2^3\}) = s_3$	$\delta(s_2, \{e_1^5\}) = s_5$
$\delta(s_3, \{e_1^6\}) = s_6$	$\delta(s_4, \{e_1^7\}) = s_7$	$\delta(s_5, \{e_1^6\}) = s_6$
$\delta(s_5, \{e_1^8, e_2^8\}) = s_8$	$\delta(s_6, \{e_1^8, e_2^8\}) = s_8$	$\delta(s_6, \{e_1^9\}) = s_9$
$\delta(s_7, \{e_1^4\}) = s_4$	$\delta(s_7, \{e_1^9\}) = s_9$	$\delta(s_8, \{e_1^{10}\}) = s_{10}$
$\delta(s_8, \{e_1^{11}\}) = s_{11}$	$\delta(s_9, \{e_1^1\}) = s_1$	$\delta(s_9, \{e_1^7\}) = s_7$
$\delta(s_9, \{e_1^{11}\}) = s_{11}$	$\delta(s_9, \{e_1^{12}\}) = s_{12}$	$\delta(s_{10}, \{e_1^2\}) = s_2$
$\delta(s_{10}, \{e_1^{11}\}) = s_{11}$	$\delta(s_{11}, \{e_1^{12}\}) = s_{12}$	$\delta(s_{11}, \{e_1^{13}\}) = s_{13}$
$\delta(s_{12}, \{e_1^{13}\}) = s_{13}$		

Based on the definitions of our state model in Section 3.1, we have $S_M = (Q, \delta, s_I, s_k, M)$, where

$Q = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}\}$,

the initial state is s_1,

the final state is s_{13}, and

the transition matrix M from {5} is computed as follows:

$$\begin{array}{c|ccccccccccccc|}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} \\
\hline
s_1 & 0 & .3992 & .5988 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_2 & 0 & 0 & 0 & 0 & .99 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_3 & 0 & 0 & 0 & 0 & 0 & .9751 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_4 & 0 & 0 & 0 & 0 & 0 & 0 & .999 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_5 & 0 & 0 & 0 & 0 & 0 & .097515 & 0 & .8865 & 0 & 0 & 0 & 0 & 0 \\
s_6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .747 & .249 & 0 & 0 & 0 & 0 \\
s_7 & 0 & 0 & 0 & .196 & 0 & 0 & 0 & 0 & .78 & 0 & 0 & 0 & 0 \\
s_8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .699986 & .299994 & 0 & 0 \\
s_9 & .1425 & 0 & 0 & 0 & 0 & 0 & .198 & 0 & 0 & 0 & .475 & .1425 & 0 \\
s_{10} & 0 & .4825 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .4825 & 0 & 0 \\
s_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .388 & .582 \\
s_{12} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .98 \\
s_{13} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

$$M(1,2) = R_1 L_{1,2} P_{1,2} = 0.998 \times 1.0 \times 0.4 = 0.3992$$

$$M(1,3) = R_1 L_{1,3} L_{1,4} P_{1,3} = 0.998 \times 1.0 \times 1.0 \times 0.6 = 0.5988$$

$$M(2,5) = R_2 L_{2,6} P_{2,6} = 0.99 \times 1.0 \times 1.0 = 0.99$$

$$M(3,6) = R_3 L_{3,7} R_4 L_{4,7} P_{3,7} = 0.98 \times 1.0 \times 0.995 \times 1.0 \times 1.0 = 0.9751$$

$$M(4,7) = R_5 L_{5,8} P_{5,8} = 0.999 \times 1.0 \times 1.0 = 0.999$$

$$M(5,6) = R_6 L_{6,7} P_{6,7} = 0.985 \times 0.99 \times 0.1 = 0.097515$$

$$M(5,8) = R_6 L_{6,9} L_{6,10} P_{6,9} = 0.985 \times 1.0 \times 1.0 \times 0.9 = 0.8865$$

$$M(6,8) = R_7 L_{7,9} L_{7,10} P_{7,9} = 0.996 \times 1.0 \times 1.0 \times 0.75 = 0.747$$

$$M(6,9) = R_7 L_{7,11} P_{7,11} = 0.996 \times 1.0 \times 0.25 = 0.249$$

$$M(7,4) = L_{8,5} P_{8,5} = 0.98 \times 0.2 = 0.196$$

$M(7,9) = R_8 L_{8,11} P_{8,11} = 0.975 \times 1.0 \times 0.8 = 0.78$

$M(8,10) = (1\text{-}(1\text{-}R_9 L_{9,12})(1\text{-}R_{10} L_{10,12}))P_{9,12}$

$= (1\text{-}(1\text{-}0.99 \times 1.0)(1\text{-}0.998 \times 1.0)) \times 0.7 = 0.699986$

$M(8,11) = (1\text{-}(1\text{-}R_9 L_{9,13})(1\text{-}R_{10} L_{10,13}))P_{9,13}$

$= (1\text{-}(1\text{-}0.99 \times 1.0)(1\text{-}0.998 \times 1.0)) \times 0.3 = 0.299994$

$M(9,1) = R_{11} L_{11,1} P_{11,1} = 0.95 \times 1.0 \times 0.15 = 0.1425$

$M(9,7) = L_{11,8} P_{11,8} = 0.99 \times 0.2 = 0.198$

$M(9,11) = R_{11} L_{11,13} P_{11,13} = 0.95 \times 1.0 \times 0.5 = 0.475$

$M(9,12) = R_{11} L_{11,4} P_{11,4} = 0.95 \times 1.0 \times 0.15 = 0.1425$

$M(10,2) = R_{12} L_{12,2} P_{12,2} = 0.965 \times 1.0 \times 0.5 = 0.4825$

$M(10,11) = R_{12} L_{12,13} P_{12,13} = 0.965 \times 1.0 \times 0.5 = 0.4825$

$M(11,12) = R_{13} L_{13,14} P_{13,14} = 0.97 \times 1.0 \times 0.4 = 0.388$

$M(11,13) = R_{13} L_{13,15} P_{13,15} = 0.97 \times 1.0 \times 0.6 = 0.582$

$M(12,13) = R_{14} L_{14,15} P_{14,15} = 0.98 \times 1.0 \times 1.0 = 0.98$

$n = 13$, $|(I\text{-}M)_{n,1}| = 0.3788$, $|I\text{-}M| = 0.4285$, $R_{15} = 0.992$

$$\text{System reliability } R = (-1)^{n+1} R_{15} \frac{|(I-M)_{n,1}|}{|I-M|} = 0.8769$$

4.2 Component Sensitivity Modeling

Component sensitivity modeling is to measure the impact of the reliability of a component on the overall system reliability. The component sensitivity measure is the differentiated value of software reliability with the component reliability. The objective of this study is to demonstrate that improving a critical component in a system can have a more significant impact to the overall system reliability than the other components, even though they all have the same component reliability. In general, a component that has more interactions or interrelationships with the other components is most likely to be a critical component. The architecture-based component sensitivity model allows us to compute the sensitivities of components in heterogeneous software architecture, composed of various component interactions and system structures.

In this section, we focus only on software that follows Markov properties without considering connector reliabilities. Therefore, the transition matrix M from {5} can be simplified as below. We have $M(i,j)$ =

$$\begin{cases} 0 & \text{if } \delta(s_i, E^j) \neq s_j \\ R_i \Gamma_{ij} & \text{if } s_i \notin \xi_P \cup \xi_F \text{ and } s_j \notin \xi_S, \Gamma_{ij} = P_{ij} \\ \left(\prod_{k=a+1}^{a+p} R_k \right) \Gamma_{ij} & \text{if } s_i \in \xi_P \text{ and } c_{a+1} \text{ to } c_{a+p} \text{ are in } s_i, \Gamma_{ij} = P_{(a+1)j} \\ \left(1 - \left(\prod_{k=b+1}^{b+q} (1 - R_k) \right) \right) \Gamma_{ij} & \text{if } s_i \in \xi_F \text{ and } c_{b+1} \text{ to } c_{b+q} \text{ are in } s_i, \Gamma_{ij} = P_{(b+1)j} \\ R_i \Gamma_{ij} & \text{if } s_j \in \xi_P \cup \xi_F \text{ and } c_{c+1} \text{ to } c_{c+r} \text{ are in } s_j, \Gamma_{ij} = P_{i(c+1)} \\ \Gamma_{ij} & \text{if } s_i \in \xi_C \text{, and } s_j \in \xi_S, \Gamma_{ij} = P_{ij} \end{cases}$$

, for $1 \leq i < n$, $1 \leq j \leq n$, where $|\xi| = n$.

After the above architecture-based transition matrix M is available, the component sensitivity model can be utilized to measure the sensitivity of each individual component through the following two phases. The first phase is to compute the sensitivity of each state. The second step is to compute the sensitivity of each component, due to a state may have multiple components working simultaneously or compensating the failure of each other.

Phase 1, compute the sensitivity S_i for each state s_i:

Based on Sections 4.2 and 4.3, let $W = (I - M)$ and $E = (I - M)_{n,1}$. We use α_{ij} to represent the cofactor of $W(i, j)$ and β_{ij} to represent the cofactor of $E(i, j)$. In addition, two functions *row*(i) and *column*(j) return the mapping state s_i and the mapped state s_j of $W(i, j)$.

For $row(i) \in \xi$, $1 \leq i \leq n$

$|W| = (\theta_{1i} + \theta_{4i}) + (\theta_{2i} + \theta_{3i})R_i$, for $i = 1, \ldots, n\text{-}1$ .... {6}

$\theta_{1i} = \alpha_{ii}$,

$\theta_{2i} = - \Gamma_{ii}\alpha_{ii}$,

$\theta_{3i} = - \sum\limits_{i \neq j, j=1}^{n} \Gamma_{ij}\alpha_{ij}$, for $column(j) \notin \xi_S$

$\theta_{4i} = - \sum\limits_{i \neq j, j=1}^{n} \Gamma_{ij}\alpha_{ij}$, for $column(j) \in \xi_S$

, where θ_{1i}, θ_{2i}, θ_{3i}, and θ_{4i} are not functions of R_i

$|E| = (\sigma_{1i} + \sigma_{4i}) + (\sigma_{2i} + \sigma_{3i})R_i$, for $i = 1, \ldots, n-1$ .... {7}

$$\sigma_{1i} = \begin{cases} 0 & \text{for } i = 1 \\ \beta_{ii(i-1)} & \text{for } 1 < i \le n-1 \end{cases}$$

$$\sigma_{2i} = \begin{cases} 0 & \text{for } i = 1 \\ -\Gamma_{ii}\beta_{ii(i-1)} & \text{for } 1 < i \le n-1 \end{cases}$$

$$\sigma_{3i} = -\sum_{i \ne j, j=2}^{n} \Gamma_{ij}\beta_{i(j-1)}, \text{ for } column(j) \notin \xi_S$$

$$\sigma_{4i} = -\sum_{i \ne j, j=2}^{n} \Gamma_{ij}\beta_{ii(j-1)}, \text{ for } column(j) \in \xi_S$$

, where σ_{1i}, σ_{2i}, σ_{3i} and σ_{4i} are not functions of R_i.

From {B2} in Appendix B, $R = (-1)^{n+1}R_n\dfrac{(\sigma_{1i} + \sigma_{4i}) + (\sigma_{2i} + \sigma_{3i})R_i}{(\theta_{1i} + \theta_{4i}) + (\theta_{2i} + \theta_{3i})R_i}$, for $i = 1, \ldots, n-1$

$$S_i = \frac{\partial R}{\partial R_i} = (-1)^{n+1} R_n \frac{(\theta_{1i} + \theta_{4i})(\sigma_{2i} + \sigma_{3i}) - (\theta_{2i} + \theta_{3i})(\sigma_{1i} + \sigma_{4i})}{((\theta_{1i} + \theta_{4i}) + (\theta_{2i} + \theta_{3i})R_i)^2}, \text{ for } i = 1, \ldots, n-1 \ldots \{8\}$$

$$S_n = \frac{\partial R}{\partial R_n} = (-1)^{n+1} \frac{(\sigma_{1i} + \sigma_{4i}) + (\sigma_{2i} + \sigma_{3i})R_i}{(\theta_{1i} + \theta_{4i}) + (\theta_{2i} + \theta_{3i})R_i} \ldots\ldots \{9\}$$

Phase 2, compute the sensitivity for each component:

Case 1: for components in batch-sequential and call-and-return styles.

In batch-sequential and client-server styles, only one component is executing in a state; thus the sensitivity obtained of this state is the sensitivity of the component.

Case 2: for components in parallel/pipe-filter styles.

Assume that there are k parallel components $C = \{c_1, c_2, \ldots, c_k\}$ in state s_i and each component c_l in C has component reliability R_l. From phase 1, we obtained that the sensitivity for state s_i is S_i. Consequently, the sensitivity of parallel component c_l in C is equal to $\left(\prod_{j=1,j\neq l}^{k} R_j\right) S_i$ which is described as follows:

$$R = (-1)^{n+1} R_n \frac{(\sigma_{1i}+\sigma_{4i})+(\sigma_{2i}+\sigma_{3i})\left(\prod_{l=1}^{k} R_l\right)}{(\theta_{1i}+\theta_{4i})+(\theta_{2i}+\theta_{3i})\left(\prod_{l=1}^{k} R_l\right)}$$

$$S_{c_l} = \frac{\partial R}{\partial R_l} = (-1)^{n+1} R_n \frac{(\theta_{1i}+\theta_{4i})(\sigma_{2i}+\sigma_{3i})-(\theta_{2i}+\theta_{3i})(\sigma_{1i}+\sigma_{4i})}{((\theta_{1i}+\theta_{4i})+(\theta_{2i}+\theta_{3i})R_l)^2}\left(\prod_{j=1,j\neq l}^{k} R_j\right) = \left(\prod_{j=1,j\neq l}^{k} R_j\right) S_i \ldots\ldots\ldots\ldots\ldots \{10\}$$

Case 3: for components in fault-tolerant styles.

Assume that there are k fault-tolerant components $C = \{c_1, c_2, \ldots, c_k\}$ in state s_i and each component c_l in C has component reliability R_l. From phase 1, we obtained that the sensitivity for state s_i is S_i. The sensitivity of fault-tolerant component c_l in C is equal to $\left(\prod_{j=1,j\neq l}^{k} (1 - R_j)\right) S_i$ illustrated as follows:

$$R = (-1)^{n+1} R_n \frac{(\sigma_{1i}+\sigma_{4i})+(\sigma_{2i}+\sigma_{3i})\left(1-\prod_{l=1}^{k}(1-R_l)\right)}{(\theta_{1i}+\theta_{4i})+(\theta_{2i}+\theta_{3i})\left(1-\prod_{l=1}^{k}(1-R_l)\right)}$$

$$S_{c_l} = \frac{\partial R}{\partial R_l} = (-1)^{n+1} R_n \frac{(\theta_{1i}+\theta_{4i})(\sigma_{2i}+\sigma_{3i})-(\theta_{2i}+\theta_{3i})(\sigma_{1i}+\sigma_{4i})}{((\theta_{1i}+\theta_{4i})+(\theta_{2i}+\theta_{3i})R_l)^2}\left(1-\prod_{j=1,j\neq l}^{k}(1-R_j)\right)$$

$$=\left(1-\prod_{j=1,j\neq l}^{k}(1-R_j)\right)S_i \quad \ldots\ldots \{11\}$$

4.2.1 An Example

We use the same example in Section 4.1.4 to calculate the sensitivity of each component.

Phase 1, compute the sensitivity S_i for each state s_i:

From the state model, we obtain that:

ξ_B	$\{s_1, s_2, s_5, s_6, s_9, s_{10}, s_{11}, s_{12}, s_{13}\}$
ξ_P	$\{s_3\}$
ξ_F	$\{s_8\}$
ξ_C	$\{s_9, s_7\}$
ξ_S	$\{s_7, s_4\}$

Note that the component in state s_7 can be a caller as well as be a callee. In addition, the component in state s_9 is not only a component in batch-sequential style, but also a component as a caller. By {8} and {9}, the sensitivity of each state is shown:

States	Sensitivity	States	Sensitivity
s_1	0.9063	s_2	0.7403
s_3	0.5548	s_4	0.0092
s_5	0.7441	s_6	0.6162
s_7	0.0376	s_8	1.1227
s_9	0.1593	s_{10}	0.7962
s_{11}	0.8752	s_{12}	0.3721
s_{13}	0.8852		

Phase 2: compute the sensitivity of each component:

Case 1: for components in batch-sequential and call-and-return styles.

Components c_1, c_2, c_6, c_7, c_{11}, c_{12}, c_{13}, c_{14} and c_{15} are in batch-sequential style and components c_5, c_8 and c_{11} are in the client-server style. Here, component c_{11} belongs to both batch-sequential and client-server styles. Since these components are one-to-one mapping to a state, the state sensitivities are thus the component sensitivities.

Case 2: for parallel components c_3 and c_4.

Components c_3 and c_4 are executing in parallel. From {10}, the sensitivity for component c_3 is $s_3 \times R_4 = 0.5548 \times 0.995 = 0.5520$, and the sensitivity for component c_4 is $s_3 \times R_3 = 0.5548 \times 0.98 = 0.5437$.

Case 3: for fault-tolerant components c_9 and c_{10}.

Components c_9 and c_{10} are performing as fault-tolerant components in state s_8. From {11}, the sensitivity for component c_9 is $s_8 \times (1 - R_{10}) = 1.1227 \times 0.002 = 0.00224$. Similarly, the sensitivity for component c_{10} is $s_8 \times (1 - R_9) = 1.1227 \times 0.01 = 0.011227$.

Let $c_i > c_j$ mean that component c_i should be improved before c_j. The reliability improving sequence is shown as

$$c_1 > c_{15} > c_{13} > c_{12} > c_6 > c_2 > c_7 > c_3 > c_4 > c_{14} > c_{11} > c_8 > c_{10} > c_5 > c_9$$

Although the reliabilities of components c_1 and c_{10} are identical, the sensitivity results show that system reliability improvement by component c_1 is about 81 times than that by c_{10}. Thus, sensitivity analysis provides an admirable cost effective solution.

4.2.2 Discussion

The example of architecture-based sensitivity modeling in Section 4.2.1 depicts the reliability impact of each component on the overall system reliability. Therefore, the modeling is a cost effective solution to assist in allocating resources and effort.

However, the sensitivity modeling relies on the differentiation to be carried out. Unfortunately, it is not always possible, especially when system structures are rather complex, and connector reliabilities are also taken into account. In the next chapter, we will introduce the modeling of system behaviors and execution history. The result is that a component can use multiple states, which makes the previous formal sensitivity modeling infeasible. Therefore, we will discuss an informal approach to support the reliability impact of each component on overall system reliability, in order to suggest the improvement sequence for software components.

From Sections 4.1.1 to 4.1.5, we know that once the transition matrix M is constructed, we can utilize formula {B2} from Appendix B to compute software reliability. As we know, the reliability of a component ranges from 0 to 1. As long as the reliability of a component is less than 1, there is a chance to further improve its reliability. Therefore, an informal approach can be practiced as follows:

1. From a set of components C, choose $C' = \{c_i \mid$ for each $c_i \in C$, component reliability $R_i < 1\}$.

2. Decide a small real number Δr, $0 < \Delta r < 1$, so that for each element c_i in C', $(R_i + \Delta r) \leq 1$.

3. For each element c_i, increase its component reliability from R_i to $R_i+\Delta r$, modify the related entries of transition matrix M, and compute new system reliability from R to R_i'. Therefore, $(R_i'-R)$ is the increment of system reliability by improving the reliability of component c_i with Δr.

4. Sort the values of system reliability increment $(R_i' - R)$ for each component c_i in a decreasing order. Therefore, this sequence prioritizes the component improvement order. After improving the software, we yield a new system reliability R.

5. Assign $C = C'$, and continue the first step, if system reliability is still below expectations.

4.3 Experiments and a Case Study

Our architecture-based reliability model takes system structures into account based on the architectural-style point of view. Several simulations were conducted with random generated architectures to validate the modeling of one single style as well as heterogeneous styles. In addition, we conducted a case study to investigate the feasibility and limitations of this architecture-based reliability model on a real system.

4.3.1 Experiments

For the simulations, a number of architectures were randomly generated with the reliabilities of components and connectors assigned a random value within a specified range. The transition probability is assigned 1 for components that have only one immediate subsequent component. Otherwise, the transition probabilities are randomly generated with the sum equal to 1 for each component that has branches to multiple immediate subsequent components. These generated architectures encompassed software with only one single style or a heterogeneous style, based on the styles discussed in Section 4.1. We would like to confirm that different architecture compositions could still match with our model results. Furthermore, we wanted to observe whether the total number of components and different reliability levels of the components have effects on the correctness of our model.

To obtain the reliability measure of the simulation result, we store the number of correct executions out of the total number of executions. The value is converged when the standard deviation is lower than 0.001 with 95% confidence interval. The final simulation result is the average of 1000 converged values. The following table demonstrates the difference between the simulation results and our model results for styles with different numbers of components 15, 30, 50, and 100.

	Call-and-Return	Fault-Tolerance	Parallel	Batch-Sequential	Heterogeneous
Simulation - 15	0.891871	0.889854	0.876161	0.889705	0.8780822
Model - 15	0.891987	0.889890	0.876141	0.889913	0.878139
Difference	-0.000116	-0.000036	0.000020	-0.000208	-0.000057

Simulation - 30	0.963351	0.961882	0.960466	0.961898	0.9626
Model - 30	0.963635	0.961885	0.960477	0.961889	0.962687
Difference	-0.000284	-0.000003	-0.000011	0.000009	-0.000087
Simulation - 50	0.904549	0.898786	0.896218	0.898943	0.902644
Model - 50	0.906981	0.901081	0.897095	0.899833	0.902773
Difference	-0.002432	-0.002295	-0.000877	-0.000890	-0.000129
Simulation - 100	0.762556	0.760153	0.740725	0.760641	0.74249
Model - 100	0.764552	0.760144	0.740383	0.760421	0.744197
Difference	-0.001996	0.000009	0.000342	0.000220	-0.001707

Table III: Simulations as validation to our model results

In addition to those randomly generated architectures, a subsystem of an industrial stock system, which evaluates the trend of the stock market and generates evaluation reports, was studied. This subsystem is composed of 23 components with four architectural styles. Five components operate as server components, providing utility and computing functions. Three components act as fault tolerance, responding to the requesting data. Two components function concurrently to align data output to the screen and to generate hard copies. This realized structure is then simulated to compare the result with our architecture-based reliability model. The result of this stock subsystem is shown on the last column, contrasting to those randomly generated heterogeneous architectures in the previous table.

Heterogeneous	15	30	50	100	Stock
Simulation	0.8780822	0.9626	0.902644	0.74249	0.740419
Model	0.878139	0.962687	0.902773	0.744197	0.732226
Difference	-0.000057	-0.000087	-0.000129	-0.001707	0.008193

Table IV: Validation results for a stock system and heterogeneous architectures

Comparing the simulation with our model results, we observed that there is no significant difference for each specific style or for the increment on the total number of components. In addition, the results also match pretty well with high reliability software as well as low reliability software. The biggest difference is 0.008193 and the smallest difference is 0.000009. From statistical analysis, our model results achieve a 99% confidence interval to the simulation results.

4.3.2 A Case Study

This empirical study was conducted on an industrial real time component-based system, which has been used by more than 100 companies and 4000 individual users over the past two years. This system provides a set of statistical models to help traders and fund managers analyze the stock market's historical data and catch the future movement.

The system is composed of several sub-units including a data unit, business rule unit, utility unit, and presentation unit. These units serve as the mathematical libraries and were implemented using C and C++. In this study, we focused on the data unit, which contains 54 classes, 13,846 lines of code, and 921 functions. It has a total of 15 components embedded with three architectural styles, batch-sequential, parallel, and call-and-return. The database components run concurrently with the evaluation components so that modeling and data retrieval can operate simultaneously. Two components, Calculator and Matrix, serve as server components to provide complex mathematics calculations for the client components. The other components run in the sequential manner, with looping and branching conditions.

To measure the system reliability, we used the test pool of the system provided by the Quality Assurance team and ran 13,596 test inputs. Among these, we observed 121 failures and obtained system reliability of 0.9911.

To apply our model, the transition probabilities between components were collected from those 13,596 test inputs. For the transition probability between two components c_i and c_j, the value is calculated as the number of transitions from c_i to c_j over the total number of transitions from c_i to all its succeeding components. To measure the component reliability, for each component we use data recorded by the QA team during unit testing and compute the number of successful executions without crash over the total number of executions. The number of testing inputs for each component can be different depending on the component complexity. We constructed the state model of the system using the methodologies described in 4.1.1 to 4.1.5 and then utilized a Markov model to compute the system reliability using formula {B2} from Appendix B.

The system reliability computed from the model is 0.994001. We notice that the difference is approximately 0.003 between the reliability observed and the reliability computed from the model. The difference is caused by the probabilistic characteristics of the Markov model, without taking into account the deterministic behaviors such as component c_i always first calls or transits to c_j and then calls or transits to c_k. To tackle this situation, more efforts are required to refine the state model, which will be discussed in the next chapter.

Chapter 5

Deterministic Software Behaviors and Execution History Modeling

A number of Markov-based reliability models [14,21,31,48] have been proposed for measuring software reliability. Although Markov-based modeling does not depend on test data and can accommodate software changes, these reliability models are subject to an assumption that the control transfers among software components follow a Markov process. This implies that the next component to be executed will depend probabilistically on the present component alone and is independent of execution history. Therefore, it is often argued that many types of software do not fully satisfy the Markov properties, thus limiting the applicability of the Markov-based models.

Our approach aims at broadening the application domain of traditional Markov-based reliability models by developing state models that preserve the Markov properties, yet carefully relax the limitations of traditional models. We developed a context free grammar to capture the execution sequence among components, and the derivation results provide a guideline for generating new states. These states capture the execution history so that the history-independent Markov models can still be utilized to model history-dependent software behaviors. Consequently, our state models not only satisfy the Markov properties, but also capture all different execution scenarios of software; i.e., deterministic software behaviors, probabilistic software behaviors, and execution history are taken into account by elaborating all of the possible execution paths to address different outcome.

5.1 Limitations of Traditional Markov-Based Reliability Modeling

Traditional Markov-based reliability models are suitable for modeling probabilistic software behaviors due to the properties of a stochastic matrix. In this section, we describe the problems that occur when deterministic software behaviors, hybrid behaviors, and execution history are considered, based on traditional Markov-based reliability modeling.

5.1.1 Deterministic Software Behaviors

The first problem is the difficulty in modeling software with deterministic behaviors, in which the inter-component control transfers are predefined, i.e., not a probabilistic random event. A deterministic software behavior means that a chain of executions has the transition probability equal to 1 between every two consecutive components. The deterministic execution chain has either one of the following properties:

1. independent of execution history, if each component in the chain is executed exactly once.

2. dependent on execution history, if at least one component is executed more than once.

The first case is not only a deterministic process, but also a Markov process, with all its transition probabilities equal to 1. This can be modeled using traditional approaches. In the second case, there exists a component that has more than one immediate successor or predecessor. Therefore, the transition from this component to its next component depends on its predecessors, i.e., execution history. Furthermore, due to the property of deterministic processes, the sum of transition probabilities from one component to all its immediate successors is greater than 1. This violates a Markov property that the sum should be equal to 1; thus this chain of executions is not a Markov process. The modeling of a deterministic and history-dependent chain, an unresolved and challenging issue, is the theme of this chapter.

If a chain is independent of execution history, despite when and where a component is executed, the executions of the same component always uses the same state. Thus, the total number of states for a Markov process is the same as the number of components; regardless of how many times each component is executed. This is because the transitions from one state to all of its possible next states are simply assigned with probabilities without considering that in certain situations a state can only transit to some of its next states. On the other hand, if a chain is dependent on execution history, this implies that multiple states are needed to reflect the impact received from the component's predecessors. Since a component may be present several times in a chain, each presence requires a distinct index to distinguish the consequences of the impact of its execution history; thus multiple states must be used to represent various executions of this component.

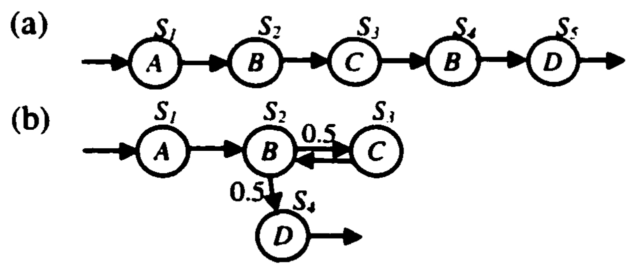


Figure 7: (a) Deterministic transition diagram and (b) probabilistic transition diagram for component running sequence →*A*→*B*→*C*→*B*→*D*→

The following example demonstrates the difficulty of modeling deterministic behaviors by using traditional Markov-based models. A deterministic process, shown as →*A*→*B*→*C*→*B*→*D*→, has transition probability 1 between every two consecutive components, in which component *B* is executed twice. The first *B* receives the control from *A*, and transfers to *C*. The second *B* receives the control from *C*, and transfers to *D*. Component *B* transferring its control to *C* or *D* does not involve uncertainties. Instead, the transition is predictable from its predecessors. Thus, component *B* is dependent on execution history. Therefore, modeling two states S_2 and S_4 for component *B* is necessary to consider deterministic behaviors and execution history, as shown in Figure 7(a).

In Figure 7(b), we demonstrate a counter example by intentionally modeling the process as a Markov process. We assign 0.5 transition probability for both *B* to *C* and *B* to *D* to satisfy Markov properties, because they have the same number of executions. To demonstrate the potential reliability difference, we assume that all the components have the same reliability *r*. In Figure 7(a) we yield software reliability r^5, which is the product of the reliabilities of the execution components as well as the transition probabilities that are all equal to 1. Figure 7(b) has a result of reliability $r^3/(2-r^2)$, considering all possible execution paths. As a result, Figure 7(a) and 7(b) can have significantly different reliability measures.

Deterministic software behaviors can also occur in the architectures of the call-and-return style [81], if a component calls several components in sequence. Because there is no conditional statement involved, we cannot assign transition probabilities from the caller to its called components. This style is classified into the following three basic types. With a caller component,

1. several components can be called in sequence,
2. its callee component can also call others, and
3. its callee component can be called a number of times.

Software architectures [27,61] can be the combinations of the above. In this Section, we will discuss the first and the second cases, where the third case regarding to the number of calls will be elaborated in Section 5.2.2.2 to address deterministic software behaviors and execution history for call-and-return style. The modeling differs from the aforementioned call-and-return style discussed in Section 4.1.4, which considers only probabilistic software behaviors and no execution history.

5.1.2 Non-Terminated Processes

The deterministic and history-dependent problem can further result in a non-terminating process when there are an infinite number of component executions. This scenario occurs when the software execution forms a loop and the execution of the loop cannot be terminated. For instance, if there are three components *A*, *B*, and *C*, component *A* first calls *B*, returns, and then terminates. However, before the termination, if *B* calls *C*, and then *C* also calls *B*, the software will only progress between *B* and *C* without termination. In this case, we need either *B* or *C* to have a branch to break the loop; otherwise, software will not terminate.

For large-scale software, there may exist non-terminated execution paths; however, the system can still provide services if those processes are not executed. To evaluate overall reliability of the software, it is necessary to take all the execution paths into account, including both terminating and non-terminating processes. Therefore, it is necessary to be able to model the reliability of software even if software has non-terminated execution paths. In addition, locating and fixing these non-terminating processes will improve software reliability.

5.1.3 An Infinite Number of Hybrid Processes

The last problem is to show the difficulty when software has an infinite number of hybrid processes. An infinite number of hybrid processes has an unlimited number of execution paths, and each hybrid process will terminate. A hybrid process is a process in software with probabilistic behaviors, and history-dependent deterministic behaviors all together.

This problem is similar to the previous problem of non-terminating processes, but a process has at least a branch to terminate the route of endless executions. Note that those endless executions are deterministic behaviors and history dependent, whereas a branch, a probabilistic behavior, prevents the non-termination problem.

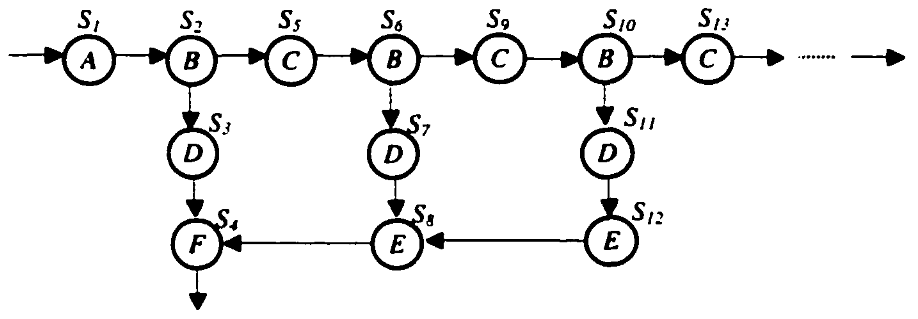


Figure 8: Infinite control transfer problem

To illustrate the above situation, we demonstrate a simple architecture of the call-and-return style, which encounters an infinite number of hybrid processes, as shown in Figure 8. This architecture has a starting component *A*, which calls component *B*, and then *F*. Component *B* calls *C* or transits to *D*. Component *C* calls *B*, and then calls *E*. Because whenever *B* calls *C*, *C* will call *B* again, an infinite number of component executions can form. The transition from *B* to *D* is a branch that prevents the non-termination. However, after finishing the execution of *D*, the control returns to *B*, returns to *C*, then executes *E*, returns to C, returns to *B*, and so on. Therefore, we understand that in each execution path component *E* must have the same number of executions as *C*, but with only one execution of component *D*. Here, we represent all these hybrid processes as $\bigcup_{n=0}^{\infty} AB(CB)^n DE^n F$, using a regular expression.

The difficulty in modeling the above scenario is to ensure that some components have the same number of executions or a certain ratio of presence in the process, such as components *C* and *E* appear the same times in the above example. To achieve that, the execution history, the deterministic behaviors, and the probabilistic behaviors all need to be taken into account.

5.2 Methodology

To tackle the aforementioned problems, addressing the execution history and software behaviors, our software reliability measurement:

1. models probabilistic, deterministic, and history-dependent software behaviors,
2. addresses the non-terminating processes that provide no service results, and
3. considers all possible execution paths, including an infinite number of hybrid processes, for the software.

Our approach is to construct a *state model* to address different software behaviors, execution history and execution paths, and then utilizes the reliability formula {B2}, from Appendix B, of a discrete-time Markov model to compute software reliability. However, this approach differs from the construction of the traditional stochastic Markov-based reliability models [21,48,58]. Traditional approaches modeled a state as an execution of a component/module, regardless of where or when the component is executed; whereas in our model, a state also depicts an index in the execution history where a component was executed. Therefore, our approach utilizes multiple states to represent the various indexes of the same component in the execution history, unlike the traditional modeling approaches that allocate only one state for one component. Accordingly, the generated states pre-capture the execution history so that the next state will depend on the current state only. This allows us to utilize the discrete time Markov models to model deterministic software behaviors and at the same time consider execution history.

For an execution path that does not terminate, our state model treats this process as a failure process and transits from the state where a component causes the non-termination to a failure state. Moreover, if a software system has an infinite number of hybrid processes, our model eliminates the requirement of an infinite number states to a finite number of states by first incorporating the construction of loops, and then modeling the structure of the loops as a binomial tree. The loops consider software behaviors and prevent the needs of history keeping, while the structure of a binomial tree covers all possible execution paths.

In the construction of the state models, we introduce a *grammar* that can be used to depict inter-component control transfers, and ultimately to derive the execution paths. The grammar comprises sets of terminals, non-terminals, and productions. A terminal represents a state at a specific index, within which a component is being executed. *Note that this component can be just one single component, or a set of components congregated into a virtual one in order to model certain types of architectural styles such as a parallel style, or fault tolerant style, etc.* A non-terminal represents the status of a component as to be executed. A production replaces the status of a component from "to be executed" to "being executed", and invokes the triggering events of this component.

Therefore, an execution path is represented as a list of terminals, derived through the productions. The grammar considers software behaviors, and addresses the execution history by allocating the same terminal in multiple indexes of a list. For example, in Figure 7(a) the grammar derives two terminals for component *B* at index two and four of a list. Depending on software, the grammar derives either a finite number or an infinite number of execution paths.

Modeling an infinite number of execution paths using a finite number of states requires a transformation scheme in order to utilize the Markov models. This scheme makes use of the structure of binomial trees, and a constructed loop derived from the grammar to model components that cause an infinite number of hybrid processes. In our model, each node in a binomial tree is a loop with an identical list of terminals and non-terminals. Each edge in the tree is a transition to and back from a designated loop. The structure of the binomial trees facilitates the modeling of all possible execution paths, by allowing the transitions among any loops within a tree. The loops not only reduce infinite states to finite states, but also ensure a correct number of component executions without the needs of history keeping.

For all of the execution paths, some may not terminate. Our software reliability measurement considers these execution paths as unreliable. Thus, in our grammar we introduce a terminal *err*, which is derived from a non-terminal that produces no terminal strings. Whenever the terminal *err* is reached, the state transition moves to a failure state in the state model.

5.2.1 Grammar

To construct the state model for both probabilistic and deterministic system behaviors, we utilize the idea of context free grammar and incorporate additional definitions and notations to gather the triggering information of a component to the other components. In the following,

- A *terminal* represents a state of the execution of a software component or the encountering of an error.
- A *non-terminal* is an intermediate symbol, corresponding to an upcoming event.
- A *production* replaces an undergoing non-terminal with terminals and non-terminals.
- A *starting symbol* is the starting point of the derivations.

The grammar $G = (T, N, P, S)$, for building a state model, consists of the following items:

1. A finite set of *terminals* $T = C \cup E$, $C = \{c_i \mid 1 \leq i \leq n\}$ and $E = \{err\}$; c_i is a software component of a system and *err* represents an error.
2. A finite set of different, intermediate symbols, called the *non-terminals* $N = \{C_i \mid 1 \leq i \leq n\}$, in which $N^1 = \{C_j \mid C_j$ derives no terminal string over C, $C_j \in N\}$.
3. A *start symbol* $S = C_1$ that starts all derivations.
4. P, a finite set of *productions*, where $P = P^1 \cup P^2 \cup P^3$.

5. P^1, a finite set of *productions* of the form $C_i \rightarrow c_i X_1 \ldots X_m$, where $C_i, X_j \in N - N^1$, $1 \le j \le m, m \ge 0$.

6. P^2, a finite set of *productions* of the form $C_i \rightarrow c_i X_1 \ldots X_k$, where $X_k \in N^1$, $C_i, X_j \in N - N^1$, $1 \le j \le k\text{-}1 < m, k \ge 1$.

7. P^3, a finite set of *productions* of the form $C_i \rightarrow err$, where $C_i \in N^1$.

Through grammar derivations of productions, we can build a state model, addressing the problems of history-dependent deterministic software behaviors, non-terminating processes, and an infinite number of hybrid processes.

5.2.2 Deterministic Software Behaviors

Unlike a probabilistic behavior, a component in a deterministic process is definite as to the next step or next few steps of its execution. Therefore, we need to address the following two issues in order to yield a correct reliability measurement. The first is to maintain the components' execution paths, which can result in a component locating at multiple different states in our state model for the same path. The second is to make sure the total number of executions for each component at each state is correctly modeled.

5.2.2.1 Maintaining Execution Paths

To maintain all possible components' execution paths for the deterministic software behaviors, we make use of the finite set of productions P^1 defined in our grammar. This set of production rules can be decomposed into three typical scenarios as follows:

- $C_i \rightarrow c_iX_1...X_m$, component c_i calls m components in sequence for services, or causes a series of transitions to m components, with $m \geq 2$.
- $C_i \rightarrow c_iC_j$, component c_i calls only one component c_j or transfers its control to component c_j, with $m = 1$.
- $C_i \rightarrow c_i$, component c_i does not call or transit to any component, with $m = 0$.

The first scenario is that a component transits to or calls another component, the second scenario is that a component transits to or calls several other components in sequence, and the last scenario is a component transits to or calls no other components. In addition, when a non-terminal C_i appears on the left hand side of multiple productions, an "*or*" statement, denoted as "|", is used to distinguish the possible selections. For example, component c_1 calls c_2 and then calls either c_3 or c_4. We utilize this "|" notation and express the example as $C_1 \rightarrow c_1C_2C_3 \mid c_1C_2C_4$ to manifest the possible choices.

A production rule records a component's invocation or triggering details to other components. Whenever carrying out a derivation, a non-terminal is replaced with terminals and non-terminals, meaning that an execution path is derived on step further. A derived terminal, excluding *err*, implies a state with a component to be activated. The total number of executions for each component at each state will be discussed in Section 5.2.2.2.

In our state model, a list of terminals, derived from the production rules, implies the execution path of the components in a list of states. Each terminal is allocated to a state and the total number of states can be more than the total number of components.

Typically, we are interested in deriving all possible execution paths as lists of terminals, which we can utilize to calculate software reliability. Therefore, we need to discuss the relationships among an execution path, a list of terminals, and the computation of reliability.

For a sequential execution path $\rightarrow c_1 \rightarrow c_2 \rightarrow \ldots \rightarrow c_k \rightarrow$, the derivations obtain k terminals with $S \rightarrow c_1 c_2 \ldots\ c_k$, due to $C_i \rightarrow c_i C_{i+1}$, $1 \leq i \leq k\text{-}1$ and $C_k \rightarrow c_k$. Following conditional probability, let $P(c_i)$ be the possibility of successful execution of component c_i, and $P(c_i|c_{i-1})$ be the possibility of successful execution of component c_i under the condition that c_{i-1} was executed successfully. Therefore, the reliability of this process is equal to either $P(c_1)P(c_2)\ldots P(c_k)$ if components are independent of each other, or $P(c_1)P(c_2|c_1)\ldots P(c_k|c_{k-1})$ if the reliability of c_i is dependent upon c_{i-1}, $2 \leq i \leq k$. Therefore, replacing each terminal in the list with a mapped component reliability, and multiplying them together yields the reliability of this execution path. For example, replace c_1 with $P(c_1)$, ..., and c_k with $P(c_k)$ or $P(c_k|c_{k-1})$.

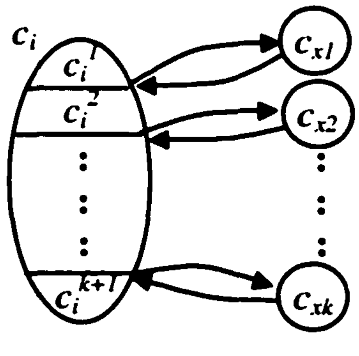


Figure 9: Component c_i calls k components $c_{x1}, c_{x2}, \ldots, c_{xk}$

Discussing deterministic behaviors of the call-and-return style, we assume a component c_i calls k components c_{x1}, c_{x2},..., c_{xk} in sequence. For simplicity, we assume that these k components will not call other components. Conceptually, component c_i is only executed once, after finishing calling those k components c_{x1}, $c_{x2}, \ldots, c_{xk}$ in sequence, even though some callees were actually called many times.

To depict into details, we assume that each component consists of a list of instructions. For generality, in this section we consider all the callees are called once and leave the discussion of the number of calls to Section 5.2.2.2. We first decompose component c_i based on the list of instructions into k+1 sections $c_i^1, c_i^2, \ldots, c_i^{k+1}$, as shown in Figure 9. Since these sections belong to the same component, we consider their dependency. Thus, the reliability of component c_i is equal to

$$P(c_i) = P(c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1})$$

$$=P(c_i^1)P(c_i^2|c_i^1)P(c_i^3|c_i^1 \cap c_i^2)\ldots P(c_i^{k+1}|c_i^1 \cap c_i^2 \cap \ldots \cap c_i^k).$$

The execution path is shown as $\rightarrow c_i^1 \rightarrow c_{x1} \rightarrow c_i^2 \rightarrow \ldots \rightarrow c_{xk} \rightarrow c_i^{k+1} \rightarrow$. The derivations yield $S \rightarrow c_i c_{x1} \ldots c_{xk}$ following $C_i \rightarrow c_i X_1 \ldots X_k$, and $X_i \rightarrow c_{xi}$, $1 \leq i \leq k$. The reliability of this execution path is

$$P(c_i^1)P(c_{x1}|c_i^1)P(c_i^2|c_i^1 \cap c_{x1})\ldots P(c_i^{k+1}|c_i^1 \cap c_{x1} \cap c_i^2 \cap \ldots \cap c_{xk})$$

$$= P(c_i^1 \cap c_{x1} \cap c_i^2 \cap \ldots \cap c_{xk} \cap c_i^{k+1})$$

$$= P(c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1} \cap c_{x1} \cap \ldots \cap c_{xk})$$

$$=\underline{P(c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1})}P(c_{x1}|c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1})P(c_{x2}|c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1} \cap c_{x1})\ldots P(c_{xk}|c_i^1 \cap c_i^2 \cap \ldots \cap c_i^{k+1} \cap c_{x1} \cap \ldots \cap c_{xk})$$

$= \underline{P(c_i)}\ P(c_{x1})\ P(c_{x2})\ldots P(c_{xk})$, if components are independent

or

$=\underline{P(c_i)}P(c_{x1}|c_i^1)P(c_{x2}|c_i^2)\ldots P(c_{xk}|c_i^k)$, where c_{xj} depends on c_i^j, $1 \leq j \leq k$

$=\underline{P(c_i)}P(c_{x1}|c_i)P(c_{x2}|c_i)\ldots P(c_{xk}|c_i)$, considering a component instead of a section as a basic unit.

Similarly, we calculate the reliability of this call-and-return execution path by replacing each terminal in the list with a mapped component reliability, such as c_1 with $P(c_1)$,..., c_{xk} with $P(c_{xk})$, or $P(c_{xk}|c_i)$ if dependent, and multiplying the component reliabilities together to yield the reliability of this execution path.

5.2.2.2 The Modeling of the Number of Calls to a Component

In Section 5.2.2.1, we derive an execution path as a list of terminals for deterministic software behaviors, based on the derivations from the production rule set P^1 of the grammar G. Each terminal represents a state with a component to be executed in that state. The terminals in a list however do not manifest the number of executions that will be carried out for the components. This information is collected based on the design information.

As a component can be called a number of times, we classify the situation into the following three scenarios:

- A component is called exactly once.
- A component is called exactly n times.

- A component is called an indefinite number of times.

Assuming three components c_1, c_2, and c_3 with component reliabilities R_1, R_2 and R_3, respectively, component c_1 calls c_2 a number of times, then calls c_3 once, and finally terminates. Applying the above three scenarios to the number of calls, we yield the same derivation results $c_1c_2c_3$ from our grammar G, as $S \rightarrow C_1 \rightarrow c_1C_2C_3 \rightarrow c_1c_2c_3$, where $C_2 \rightarrow c_2$ and $C_3 \rightarrow c_3$ are because both c_2 and c_3 call no component. The following demonstrates the construction of our state model for each scenario:

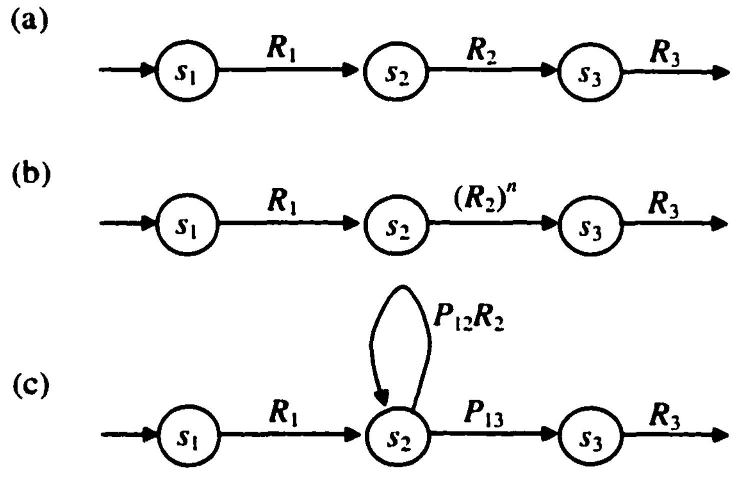


Figure 10: Three types of calling scenarios

Scenario 1, component c_1 calls component c_2 exactly once and then calls c_3 once:

In this scenario, components c_1, c_2, and c_3 are all executed exactly once. Therefore, system reliability should yield $R_1R_2R_3$. The modeling of this scenario has been addressed in the deterministic software behaviors, discussed in Section 5.2.2. With $s_i = (E^i, C^i)$ and state model S_M defined in Section 3.1, this scenario has three states $s_1 = (\{e_1^1\}, \{c_1\})$, $s_2 = (\{e_1^2\}, \{c_2\})$ and $s_3 = (\{e_1^3\}, \{c_3\})$, and two possible state transitions $\delta(s_1, \{e_1^2\}) = s_2$ and

$\delta(s_2, \{e_1^3\}) = s_3$. When the transition reaches state s_2, event e_1^2 is to activate component c_2. Similarly, when the transition reaches state s_3, event e_1^3 is to activate component c_3. We construct the state model $S_M = (Q, \delta, s_1, s_3, M)$, where $Q = \{s_1, s_2, s_3\}$.

Figure 10(a) is the state diagram of this scenario, where s_1, s_2, and s_3 are the mapping states to components c_1, c_2 and c_3. A 3×3 matrix M is constructed as below.

$$\begin{cases} M(1,2) = R_1, \text{ state } s_1 \text{ reaches state } s_2 \\ M(2,3) = R_2, \text{ state } s_2 \text{ reaches state } s_3, \text{ for } 1 \le i, j \le 3 \\ M(i,j) = 0, \text{ otherwise} \end{cases} \qquad \text{.......} \{12\}$$

Scenario 2, component c_1 calls component c_2 exactly n times and then calls c_3 once:

In this scenario, we understand that both components c_1 and c_3 are only executed once, but component c_2 is executed exactly n times. Therefore, system reliability should yield $R_1(R_2)^n R_3$, where $(R_2)^n$ considers those n times of execution of component c_2. This scenario is almost identical to scenario 1, except that we have to consider component c_2 is executed n times not just once.

Figure 10(b) is the state diagram of this scenario, where s_1, s_2, and s_3 are the mapping states to components c_1, c_2 and c_3. A 3×3 matrix M is constructed as below. Note that $M(2,3)$ has entry value $(R_2)^n$ to address the n times of executions of component c_2.

$$\begin{cases} M(1,2) = R_1, \text{ state } s_1 \text{ reaches state } s_2 \\ M(2,3) = (R_2)^n, \text{ state } s_2 \text{ reaches state } s_3, \text{ for } 1 \le i, j \le 3 \\ M(i,j) = 0, \text{ otherwise} \end{cases} \qquad \text{.......} \{13\}$$

Scenario 3, component c_1 calls component c_2 an indefinite number of times and then calls c_3 once:

In this scenario, components c_1 and c_3 are both executed once. However, component c_1 may call component c_2 possibly 0 time to many times. Therefore, we know that component c_1 will encounter a condition to either transit to c_2 or c_3. We assume that the transition probability from c_1 to c_2 is P_{12}, and the transition probability from c_1 to c_3 is P_{13}. Therefore, system reliability should yield $R_1R_{13} / (1 - P_{12}R_2)$. Based on the definitions of $s_i = (E^i, C^i)$ and state model S_M in Section 3.1, we construct three states $s_1 = (\{e_1^1\}, \{c_1\})$, $s_2 = (\{e_1^2\}, \{c_2\})$ and $s_3 = (\{e_1^3\}, \{c_3\})$, and three possible state transitions $\delta(s_1, \{e_1^2\}) = s_2$, $\delta(s_2, \{e_1^2\}) = s_2$ and $\delta(s_2, \{e_1^3\}) = s_3$.

Typically, this problem is basically identical to the call-and-return problem in Section 4.1.4, except that we would like to consider the deterministic behaviors. Thus, we arrange the configuration into a different view as shown in Figure 10(c), which is the state diagram of this scenario, where s_1, s_2, and s_3 are the mapping states to components c_1, c_2 and c_3. A 3 × 3 matrix M is constructed as below. Note that $M(2,3)$ has entry value P_{13}, instead of R_2P_{13}, to enable component c_2 to be executed 0 time. The entry $M(2,2)$ has value $P_{12}R_2$ in a loop to address the indefinite number of executions of component c_2.

$$\begin{cases} M(1,2) = R_1, \text{ state } s_1 \text{ reaches state } s_2 \\ M(2,2) = P_{12}R_2, \text{ state } s_2 \text{ reaches state } s_2 \\ M(2,3) = P_{13}, \text{ state } s_2 \text{ reaches state } s_3 \\ M(i,j) = 0, \text{ otherwise} \end{cases}, \text{ for } 1 \le i, j \le 3 \quad \ldots\ldots\ldots\ldots \{14\}$$

5.2.3 Non-Terminating Processes

A non-terminating process, having an infinite number of component executions, does not terminate. The derivations for this process yield no terminal string. Such a process, providing no service results to a request, is considered as an unreliable process in our model.

To model, we look for the first non-terminal C_k that appears in the derivations, where $C_k \in N^1$. Because this C_k produces no terminal string, the execution path will definitely not terminate. In other words, C_k results in an unreliable service. Therefore, we replace C_k with the terminal *err*, and eliminate the derivations after C_k. The terminal *err* represents a transition to the failure state in our state model, so that the computation of software reliability measurement will not include this non-terminating process.

In details, we introduce two finite sets of productions P^2 and P^3 in the defined grammar. In general, the triggering information is modeled as a production in P^1 with the format $C_i \rightarrow c_i X_1 \ldots X_m$. However, if $\exists X_k \in N^1$, $k < m$, $X_j \in (N\text{-}N^1)$, $1 \le j < k$, we *instead* add two productions $C_i \rightarrow c_i X_1 \ldots X_k$ to P^2, and $X_k \rightarrow err$ to P^3. With P^2 and P^3, a non-terminating process now ends with a list of terminals, with one and only one terminal *err* at the end of the list. Therefore, when our state model reads in the terminal *err*, the transition goes to a failure state.

5.2.4 An Infinite Number of Hybrid Processes

Hybrid software behaviors, composed of probabilistic and history-dependent deterministic behaviors, may result in an infinite number of execution paths. In Figure 8, there exists an infinite number of execution paths, represented as $\bigcup_{n=0}^{\infty} AB(CB)^n DE^n F$, using a regular expression. In this case, the derivations will produce an unlimited number of lists of terminals. This requires an infinite number of states and hinders the use of the Markov model for software reliability measurement.

Therefore, we introduce a transformation scheme to resolve the problem of an infinite number of hybrid processes. The objective of the transformation is to:

- reduce infinite states to finite states, and
- model all possible execution paths to ensure that the reliability measurement is not over- or under-estimated.

To reduce infinite states to finite states, the idea is to model a finite-state loop, defined as *a recurrent loop*, for each branching component, defined as *a break-point component* that has at least one but not all branches causing an infinite number of hybrid processes. To model all possible execution paths, the recurrent loop is constructed based on the structure of *binomial trees* that tackles all the execution paths. A node in the binomial tree is *a set of basic clusters*, and an edge is a directed link for the transition from one set of basic clusters to another identical set of basic clusters. In the following, we will use our previous grammar to define *a break-point component* and *its set of basic clusters*.

- *A break-point component* c_i is a branching component, whose mapping non-terminal C_i appears on the left hand side of multiple productions, in which at least one but not all productions derive C_i on the right hand side, where $C_i \in N - N^1$.

- *A set of basic clusters*, for the break-point component c_i, is a set of lists yielded through the productions starting from C_i, where the derivations continue on all non-terminals except C_i, and produce at least one C_i in each list. However, for each C_j, the non-terminal of another break-point component c_j where $i \neq j$, occurs during the derivations of C_i, the production rules of C_j that produce no non-terminals C_j are chosen to continue the derivations.

A *break-point component* c_i guarantees the occurrence of an infinite number of execution paths, because C_i can derive C_i on the right hand side. However, this branching component has at least one branch to stop calling itself persistently. A basic cluster of a break-point component c_i combines a list of terminals with a number of non-terminals C_i. The derived terminals depict which components to execute and the non-terminals C_i indicate that a break-point component c_i can further derive itself a number of times. Note that our model considers the number of derived non-terminals C_i in a basic cluster as finite.

For a break-point component, our transformation scheme first adjusts each basic cluster by moving its *final* non-terminal to the end of the list. Due to no history keeping in the Markov model, this adjustment prevents history keeping and ensures that some components have the same number of executions or a certain ratio of presence in the process. Whenever the final non-terminal is reached in a basic cluster, a transition goes back to the state of the first derived terminal without generating a new state. Therefore, a loop is formed with the final non-terminal shares the same state as its first derived terminal. This sharing means that the final non-terminal now takes over the position of its own break-point component. Note that such a loop is a subset of a recurrent loop.

If we take Figure 8 for example, component *B* is a break-point component, with one branch deriving *B* itself. There is a basic cluster *bcBe* for component *B* that is derived from $B \rightarrow bC \rightarrow bcBE \rightarrow bcBe$. Note that *B* is in the list and not further processed. Following the adjustment, we move the final non-terminal to the end of the list. The basic cluster *bcBe* becomes *bceB*. Figure 8 can be transformed into Figure 11 with *b* to *B* sharing the same state. The original regular expression $\bigcup_{n=0}^{\infty} AB(CB)^n DE^n F$ becomes $AB(CBE)^* DF$.

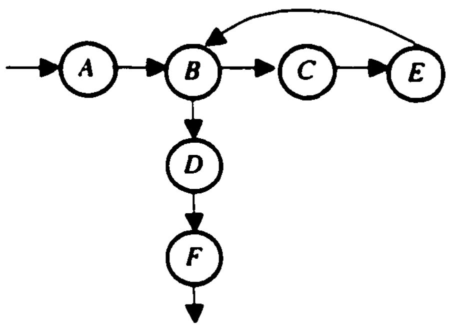


Figure 11: An infinite number of hybrid processes

The reliability of an execution path is the product of the reliabilities of the executed components, and connectors as well as the underlying transition probabilities. Based on the commutative law in mathematics, *ab* is equal to *ba* if *a* and *b* are real numbers. Because the reliabilities and probabilities are all real numbers, the adjustment will not yield a different result. Thus, we can expect the same software reliability from the two regular expressions above.

The above is a simple example and the recurrent loop is the same as its basic cluster. In the following, we will explore the modeling of a break-point component having either only one single basic cluster or having multiple basic clusters.

5.2.4.1 Single Basic Cluster

Here we model a break-point component c_i, having only one basic cluster that includes k non-terminals C_i in the list. The aforementioned adjustment to a basic cluster is our first step in building a loop, as is the case when $k = 1$ for Figure 8. For $k \geq 2$, we not only need the adjustment, but also have to address all of the possible execution paths. Since in a basic cluster each of k non-terminals C_i has the capability to go through the basic cluster again, there are k^{m-1} alternative execution paths for going through m iterations of the basic cluster, where $m \geq 1$.

Recall the construction of a loop for a basic cluster. The final non-terminal shares the same position as its first derived terminal. Since all the non-terminals are identical, for illustration we denote the invoking non-terminal as C_i, which invokes k occurrences of C_i, denoted as C_i^1, C_i^2, ..., C_i^k. Forming a loop, we have the first derived terminal c_i and C_i^k share the same position, as shown in Figure 12. After the sharing, C_i^k takes over the position of its own break-point component c_i, and can invoke k non-terminals inside its own basic cluster again.

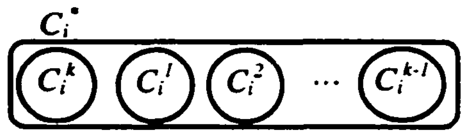


Figure 12: A starting non-terminal C_i, denoted as C_i^*, invokes k times of C_i, denoted as $C_i^1, C_i^2, \ldots, C_i^k$.

Most importantly, $C_i^1, C_i^2, \ldots, C_i^{k-1}$ can each also invoke k non-terminals, and result in an infinite number of hybrid processes. Likewise, we utilize the same basic cluster for $C_i^1, C_i^2, \ldots, C_i^{k-1}$, as shown in Figure 13. Assuming $1 \leq s \leq t \leq k$, when the C_i^t, invoked by C_i^s, invokes another k non-terminals, we can proceed the control from the loop of C_i^s to the loop of C_i^t, without violating the execution sequence, because $s \leq t$. However, this is not backward compatible when C_i^s is invoked by C_i^t, $s < t$. Because the loop of C_i^t is already behind the loop of C_i^s, the control from the C_i^t loop to the C_i^s loop subsequently must go through the loop of C_i^k.

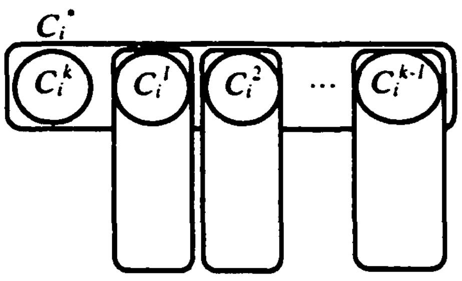


Figure 13: $C_i^1, C_i^2, \ldots, C_i^k$ can each further invoke another k non-terminals.

To resolve the backward compatibility problem, our transformation scheme utilizes the characteristics of *binomial trees*, as shown in Figure 14. As a result, the exact possible execution paths can be modeled and infinite states can be reduced to finite states. In [16], the properties of binomial tree B_m are summarized as follows:

1. there are 2^m nodes,
2. the height of the tree is m, and
3. there are exactly $\binom{m}{i}$ nodes at depth i for $i = 0, 1, \ldots, m$, and the root has degree m, which is greater than that of any other node; if the root's children are numbered from left to right by m-1, m-2,...,0, child i is the root of a subtree B_i.

The structure of a binomial tree can be concluded as a higher degree node always has links to all its lower degree nodes. The root node of tree B_m has degree m and its m-1 children have degrees m-1 to 0 from left to right. Likewise, a node with degree k, $k < m$, is the root for binomial tree B_k, whose children have degrees k-1 to 0 from left to right. This can resolve our backward compatibility problem.

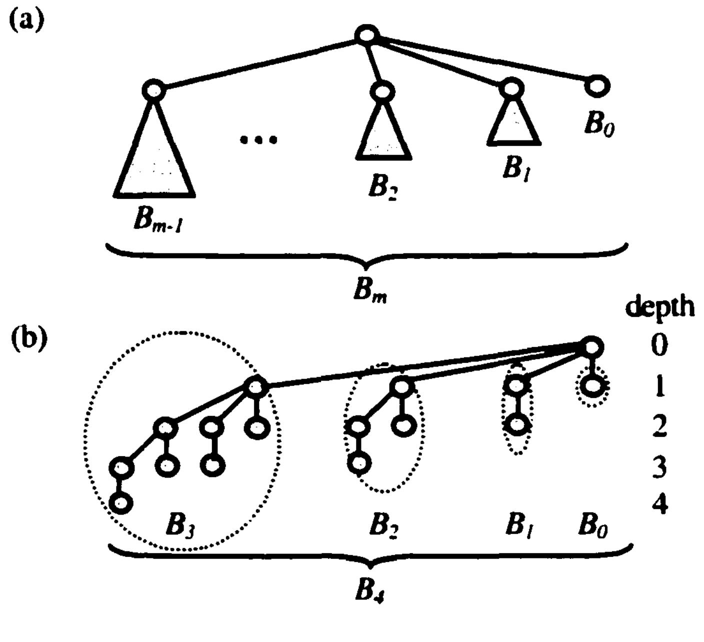


Figure 14: (a) The binomial tree B_m is represented. A triangle represents a rooted subtree. (b) Node depths in B_4 are shown.

Recall that the backward compatibility problem occurs when C_i^s is invoked by C_i^t and $s < t$. We cannot advance from the loop of C_i^t to the loop of C_i^s. To the contrary, the structure of the binomial tree allows a node with higher degree to go to all its lower degree nodes. Therefore, if we construct the structure of C_i^t with the same structure as the binomial tree B_{t-1}, we can allow C_i^s be invoked by C_i^t and resolve the problem.

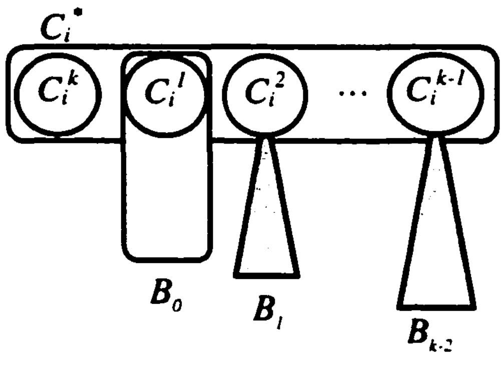


Figure 15: The transformation scheme for a break-point component C_i invoking k non-terminals C_i.

Essentially, a node in a binomial tree represents a basic cluster that forms a loop in our transformation process. When a basic cluster has k non-terminals, the binomial tree B_{i-1} is utilized to construct the structure of the ith non-terminal, $1 \leq i \leq k-1$. After the construction shown in Figure 15, we can see the structure of the kth non-terminal has the structure of the binomial tree B_{k-1}. This matches Figure 14(a). In other words, there will be 2^{k-1} of basic cluster expansion for resolving k non-terminals.

After the transformation is completed, an additional step is to resolve the non-terminal C_i into terminals inside each loop. For each C_i who will further invoke a basic cluster, we replace C_i with c_i. Otherwise, we replace C_i with the terminals derived from the productions that will not cause a basic cluster. Since $C_i \in N - N^1$, C_i can derive terminal strings.

Here we show an example where a break-point component c_2 is called by c_1. Its mapping non-terminal C_2 either go through $C_2 \rightarrow c_2c_3C_2c_4C_2c_5C_2c_6$, or follows a production $C_2 \rightarrow c_2c_7$. Note that the basic cluster of c_2 is $c_2c_3C_2c_4C_2c_5C_2c_6$. We know that the binomial tree B_2 will be utilized for the transformation. We first move the final C_2 to the end of the list and obtain $c_2c_3C_2c_4C_2c_5c_6C_2$. Figure 16(a) shows the loop, where the final non-terminal C_2 shares the same position as the invoking C_2, Figure 16(b) shows the complete transformation by using B_2 binomial tree, and Figure 17 shows the transition diagram with all terminals.

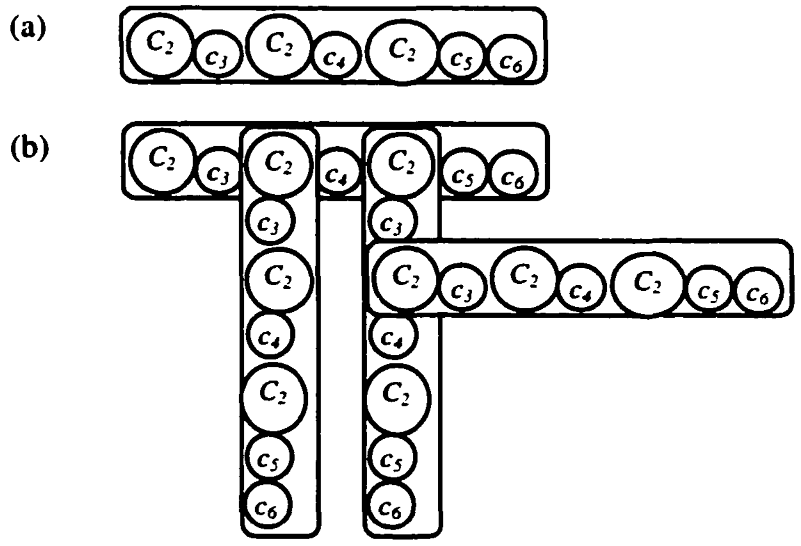


Figure 16: (a) A loop based on a basic cluster (b) The transformation by using B_2 binomial tree

Note that the modeling removes some of the branches from c_2 to c_3, we have to change those transition probabilities between components c_2 and c_7 to 1. This is easy. Suppose that component c_2 has component reliability R_2, and to c_7 has transition probability P_{27}. If we want to replace P_{27} with 1, we can replace R_2 with R_2P_{27} and achieve the same result.

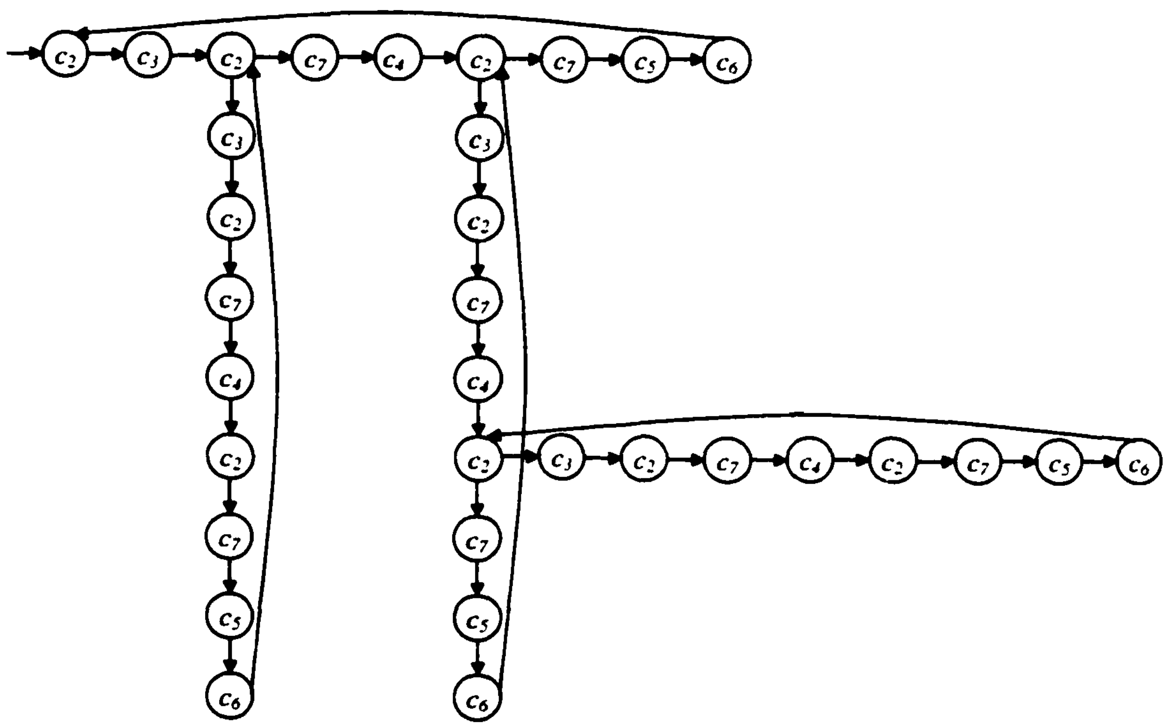


Figure 17: The transition diagram with all terminals

The above is just a single branch adjustment. Assume that there are branches removed from component c_i to its subsequent components and more than one branch is left. If the summation of the transition probabilities of these left branches is equal to P. We can adjust the reliability of c_2 as R_2P and adjust each branch by multiplying $1/P$ to its original transition probability. In this case, the sum of the transition probabilities of all the left branches is equal to 1.

5.2.4.2 Multiple Basic Clusters

Multiple basic clusters occur when a break-point component has more than one basic cluster. In this case, we have to model a set of basic clusters with more than one element for this break-point component. In the set, each basic cluster can derive a different number of non-terminals.

The modeling for each basic cluster is basically the same as the aforementioned single basic cluster model. However, to ensure that a derived non-terminal can choose an alternative basic cluster to execute, each non-terminal, which originally expanded with one basic cluster in the previous section, expands with a set of basic clusters. After the modeling of all basic clusters, the initial break-point component also expands with a set of basic clusters.

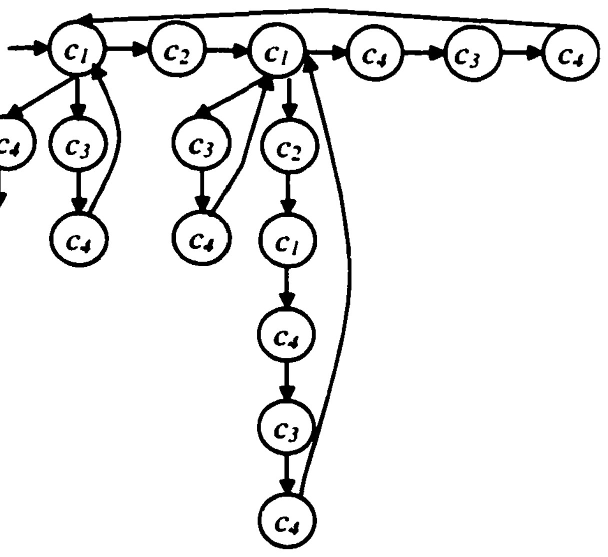


Figure 18: The break-point component c_1 has two basic clusters $c_1c_2C_1c_3C_1c_4$ and $c_1c_3C_1c_4$

Figure 18 shows an example of a break-point component, with two basic clusters, based on the following productions:

$$S \to C_1 \quad C_1 \to c_1C_4 \quad C_1 \to c_1C_2 \quad C_1 \to c_1C_3 \quad C_2 \to c_2C_1C_3 \quad C_3 \to c_3C_1C_4 \quad C_4 \to c_4$$

Based on the grammar, we yield two basic clusters $c_1c_2C_1c_3C_1c_4$ and $c_1c_3C_1c_4$ as a set of basic cluster for the break-point component c_1. For the basic clusters $c_1c_2C_1c_3C_1c_4$ and $c_1c_3C_1c_4$, we utilize binomial trees B_1 and B_0, respectively. The main difference of multiple basic clusters, comparing to one single basic cluster, is that a break-point component now expands with the complete set of basic clusters instead of just one. Such a modeling is to ensure that a break-point component, with multiple basic clusters, can transit to any basic cluster in the set.

5.3 Discussion

We develop a context free grammar and utilize the production rules to model the triggering information among software components. The derivations of the production rules produce either a finite set of terminal strings or an infinite set of terminal strings. Typically, our architecture-based reliability model makes use of the set of terminal strings to construct a state model to address execution history and then use this state model to compute software reliability.

However, if the set of terminal strings is an infinite set, we encounter a challenge to model an infinite number of states through the finite state Markov models. Therefore, we develop a technique by identifying the break-point components, and producing the set of basic clusters for each break-point component. With a minor *permutation* to construct a loop for each basic cluster, we resolve the modeling of an infinite number of execution paths utilizing binomial tree structures.

In this section, we justify that the minor *permutation* does not affect the final reliability measurement results, due to the commutative law that $ab = ba$, if a and b are numbers. We take a simple software with three production rules to demonstrate that the reliability results are identical between the mathematics computation following the grammar derivations and our modeling approach in the architecture-based software reliability model. Therefore, when software becomes more complex that inhibits the simple mathematics computation through the grammar derivations, we can still take advantage of our architecture-based reliability models to compute software reliability.

Let R_s be the reliability of component s and R_r be the reliability of component r. In this software, we have

$S \longrightarrow sSR$, transition probability P

$S \longrightarrow s$, transition probability 1-P

$R \longrightarrow r$, transition probability 1

Component s either calls s itself once and then calls component r once with a probability of P, or calls no component with a probability of 1-P. When component s is called once by itself, this called component s can further decide to call s itself once and then r once with a probability of P, or not to call any component with a probability of 1-P. Component r never calls any component with a probability of 1.

5.3.1 Mathematics Computation:

From the grammar derivations, we have the following terminal strings:

s, ssr, $sssrr$, $ssssrrr$, ...

The occurrence possibility for each string is equal to:

$(1\text{-}P)$, $P(1\text{-}P)$, $P^2(1\text{-}P)$, $P^3(1\text{-}P)$, ...

$$\begin{tikzcd}
S \arrow[r, "P"] \arrow[dr, "1-P"'] & sSR \arrow[r, "P"] \arrow[dr, "1-P"'] & ssSRR \arrow[r, "P"] \arrow[dr, "1-P"'] & sssSRRR \cdots\cdots \arrow[dr, "1-P"'] & & \\
 & s & ssR \arrow[dr, "1"'] & sssRR \arrow[dr, "1"'] & \ddots & \\
 & & & ssr & sssrR \arrow[dr, "1"'] & \\
 & & & & & sssrr
\end{tikzcd}$$

The system reliability is computed as

$R_s(1\text{-}P) + R_s^2R_rP(1\text{-}P) + R_s^3R_r^2P^2(1\text{-}P) + \ldots$

$$= \sum_{n=1}^{\infty} R_s^{\,n} R_r^{\,n-1} P^{n-1}(1-P) = R_s(1\text{-}P)\,\frac{1}{1-R_sR_rP} = \frac{R_s(1-P)}{1-R_sR_rP} \quad \ldots\ldots \{15\}$$

5.3.2 Modeling Approach:

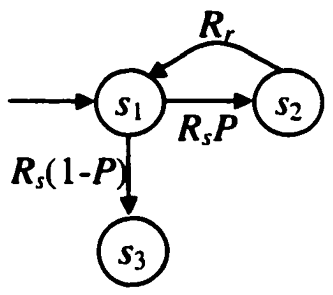
Component s is a break-point component, because it has a branch and a basic cluster srS, derived from $S \rightarrow sSR \rightarrow sSr$. We first move the last non-terminal S to the end of the list to become srS, and then combine the small s with the big S into state s_1 to form a loop, and allocate component r to state s_2. A final state s_3 is added for a virtual component f to assure that all the states in the transition matrix M are transient states, meaning $M(3,j) = 0$, $1 \le j \le 3$. The component reliability R_f for this virtual component f is assigned with 1 without affecting the reliability measurement.

$s_1 = (\{e_1^1\}, \{s\}), s_2 = (\{e_1^2\}, \{r\}), s_3 = (\{e_1^3\}, \{f\})$

$\delta(s_1, \{e_1^2\}) = s_2$, $\delta(s_2, \{e_1^1\}) = s_1$, and $\delta(s_1, \{e_1^3\}) = s_3$

$M(1,2) = R_sP$, $M(2,1) = R_r$, $M(1,3) = R_s(1\text{-}P)$, and 0's for all the other entries.

Based on the above three states, the state diagram is shown as follows:



$$T = \begin{array}{c|ccccc} & S & F & s_1 & s_2 & s_3 \\ \hline S & 1 & 0 & 0 & 0 & 0 \\ F & 0 & 1 & 0 & 0 & 0 \\ s_1 & 0 & 1-R_s & 0 & R_sP & R_s(1-P) \\ s_2 & 0 & 1-R_r & R_r & 0 & 0 \\ s_3 & 1 & 0 & 0 & 0 & 0 \end{array}, \qquad M = \begin{bmatrix} 0 & R_sP & R_s(1-P) \\ R_r & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},$$

$$I - M = \begin{bmatrix} 1 & -R_sP & -R_s(1-P) \\ -R_r & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad (I - M)_{3,1} = \begin{bmatrix} -R_sP & -R_s(1-P) \\ 1 & 0 \end{bmatrix}$$

$|I - M| = 1 - R_rR_sP$

$|(I - M)_{k,1}| = R_s\,(1\text{-}P)$

$$\text{System reliability } R = (-1)^{k+1}\frac{|(I-M)_{k,1}|}{|I-M|}R_f = (-1)^{3+1}\frac{R_s(1-P)}{1-R_rR_sP}\times 1 = \frac{R_s(1-P)}{1-R_sR_rP} \text{ ...... } \{16\}$$

From {15} and {16}, we conclude that our modeling approach yields the same result as the computation through mathematics computation.

5.4 Implementation

In Chapter IV, we proposed an architecture-based software reliability model, which takes software architecture as input and measures reliability based on the understanding of architectural styles [24,26,82]. Although the approach follows Markov properties, it assumes a software process is a Markov process. In this Chapter, we relax this assumption in order to apply the model to a wider scope of software. Therefore, this model incorporates both probabilistic and deterministic software behaviors, and takes execution history into account as well. Here we present the stepwise construction of our state model:

1. Our model takes deterministic as well as probabilistic software behaviors as inputs, where a component can call or transit to a number of components in sequence or based on condition.

   For example, there are deterministic behaviors such as the transitions from *A* to *B*, *C* to *F*, *G* in sequence, *D* to *F*, *G* in sequence, *E* to *H*, *G* to *F*, *I* in sequence, *H* to *I*, *I* to *J*, and a conditional behavior from *B* to either *C*, *D*, or *E*.

2. For generality, assuming software starts from the execution of an initial component, a grammar is developed and utilized to record the triggering and transition information into productions. When encountering conditional statements, multiple productions share the same non-terminal on the left hand

side. This is frequently seen as an "|" statement. The grammar facilitates the modeling of history-dependent deterministic software behaviors.

From the transition information shown above, assuming A is the initial component, we have the following productions:

$S \rightarrow A$ $A \rightarrow aB$ $B \rightarrow bC \mid bD \mid bE$ $C \rightarrow cFG$ $D \rightarrow dFG$

$E \rightarrow eH$ $F \rightarrow f$ $G \rightarrow gFI$ $H \rightarrow hI$ $I \rightarrow iJ$ $J \rightarrow j$

3. The derivations of the grammar can produce lists of terminals that illustrate different execution paths, which may be Markov processes, deterministic processes, or hybrid processes. A terminal means that its mapping component is being executed in a state, and the position of each terminal in the list represents an index of the transition. In our model, our focus is not just in generating all of the lists of terminals, but also in minimizing the total number of states required to construct the state model, in order to compute software reliability.

   For example, the productions, from item 2 above, yield three lists of terminals as *abcfgfij*, *abdfgfij*, and *abehij*, as shown is Figure 19 below. As we can see, these three lists share the same prefix "*ab*", which only requires two states for terminals *a* and *b*. The first and second lists also share the same postfix "*fgfij*", which only requires five states for two *f*'s, one *g*, one *i*, and one *j*. Figure 19 shows a total number of 11 states of those three lists of terminals.

Three derived lists of terminals

$$\begin{cases} a\ b\ e\ h\ i\ j \\ a\ b\ d\ f\ g\ f\ i\ j \\ a\ b\ c\ f\ g\ f\ i\ j \end{cases}$$

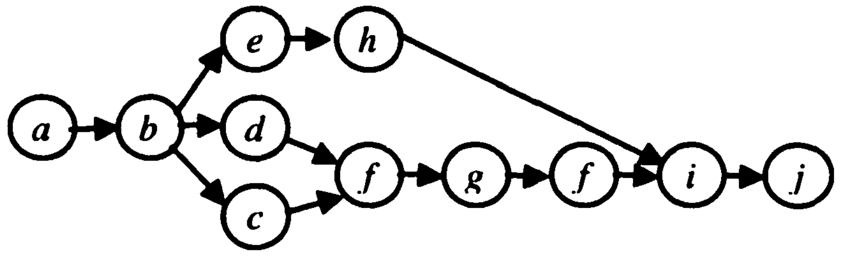


Figure 19: Terminal lists vs. a state diagram

4. The grammar assists in detecting a non-terminating process, in the situation that a non-terminal derives no terminal string. For such a non-terminal, the state model has this process transiting to the failure state.

   For example, there are two productions $A \rightarrow aBA$, and $B \rightarrow b$. We understand that A cannot derive a terminal string. Therefore, when the derivations reach A, the execution path will never terminate. For such a non-terminal, we have a production with $A \rightarrow err$. Therefore, when this non-terminal is reached, the state model has this process transiting to the failure state.

5. The state model utilizes the defined grammar to model deterministic behaviors, but is insufficient to model an infinite number of execution paths. Thus, we introduce the construction of a recurrent loop, which is composed of a number of loops constructed from a set of basic clusters. To ensure all the execution paths are taken into account, the structure of binomial trees B_{k-1} is used for a break-

point component with k basic clusters in the set. As a result, the required states are reduced from an infinite number to a finite number and at the same time all the execution paths are modeled.

5.5 An Example

A software system consists of six components c_1, c_2, c_3, c_4, c_5, and c_6. The initial component c_1 calls c_2 once, returns, then calls c_5 once, and then terminate. Component c_2 either calls c_3 once, or transits to c_6. Component c_3 calls c_2 once, returns, and then calls c_4. Component c_4 calls c_2 once, returns, and then calls c_5. The reliability R_i of each component c_i is shown below:

$$R_1 = 0.999 \quad R_2 = 1.0 \quad R_3 = 0.998 \quad R_4 = 0.997 \quad R_5 = 0.999 \quad R_6 = 1.0$$

The reliabilities of connector L_{ij} are equal to 1 except $L_{2,3}$ = 0.999. For the deterministic software behaviors, the transition probability from a state to its next state is equal to 1, except for a branching component. Therefore, we only need to take into account the transition probabilities of a branching component. In our example, component c_2 is a branching component because it transits to either c_3 or c_6. We assume that the probabilities for $P_{2,3}$ and $P_{2,6}$ are as follows:

$$P_{2,3} = 0.2 \quad P_{2,6} = 0.8$$

The following is the constructed grammar $G = (T, N, P, S)$.

$$T = C \cup E = \{c_1, c_2, c_3, c_4, c_5, c_6, err\}$$

$N = \{C_1, C_2, C_3, C_4, C_5, C_6\}$

$P = P^1 \cup P^2 \cup P^3$

$P^1 = \{C_1 \rightarrow c_1C_2C_5,\quad C_2 \rightarrow c_2C_3 \mid c_2C_6,\quad C_3 \rightarrow c_3C_2C_4,\quad C_4 \rightarrow c_4C_2C_5,$

$\quad C_5 \rightarrow c_5,\quad C_6 \rightarrow c_6\}$

$P^2 = \phi, P^3 = \phi$

$S \rightarrow C_1$

Component c_2 is a break point component, because it is a branching component and its mapping non-terminal C_2 derives C_2 itself. Note that the basic cluster of c_2 is $c_2c_3C_2c_4C_2c_5$, derived from $C_2 \rightarrow c_2C_3 \rightarrow c_2c_3C_2C_4 \rightarrow c_2c_3C_2c_4C_2C_5 \rightarrow c_2c_3C_2c_4C_2c_5$. We know that the binomial tree B_1 will be utilized for the transformation, because one C_2 can derive two non-terminals of C_2. We first move the final C_2 to the end of the list and obtain $c_2c_3C_2c_4c_5C_2$. With this basic cluster and the production set P^1, we construct the state diagram, as shown in Figure 20.

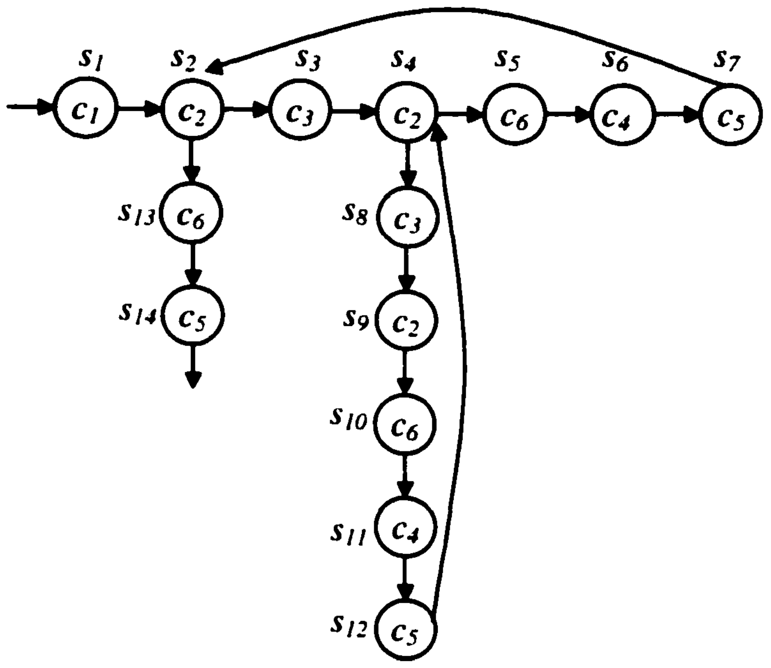


Figure 20: A state diagram for sample software with one basic cluster

The following shows the construction of our state model S_M. With $s_i = (E^i, C^i)$, and $\delta(s_i, E^j) = s_j$, this software has a total number of fourteen states from s_1 to s_{14}, as shown in Figure 20.

$s_1 = (\{e_1^1\}, \{c_1\})$	$s_2 = (\{e_1^2\}, \{c_2\})$	$s_3 = (\{e_1^3\}, \{c_3\})$
$s_4 = (\{e_1^4\}, \{c_2\})$	$s_5 = (\{e_1^5\}, \{c_6\})$	$s_6 = (\{e_1^6\}, \{c_4\})$
$s_7 = (\{e_1^7\}, \{c_5\})$	$s_8 = (\{e_1^8\}, \{c_3\})$	$s_9 = (\{e_1^9\}, \{c_2\})$
$s_{10} = (\{e_1^{10}\}, \{c_6\})$	$s_{11} = (\{e_1^{11}\}, \{c_4\})$	$s_{12} = (\{e_1^{12}\}, \{c_5\})$
$s_{13} = (\{e_1^{13}\}, \{c_6\})$	$s_{14} = (\{e_1^{14}\}, \{c_5\})$	

From the above states, the following shows 15 possible transitions between two states out of those 14 states:

$\delta(s_1, \{e_1^2\}) = s_2$	$\delta(s_2, \{e_1^3\}) = s_3$	$\delta(s_2, \{e_1^{13}\}) = s_{13}$
$\delta(s_3, \{e_1^4\}) = s_4$	$\delta(s_4, \{e_1^5\}) = s_5$	$\delta(s_4, \{e_1^8\}) = s_8$
$\delta(s_5, \{e_1^6\}) = s_6$	$\delta(s_6, \{e_1^7\}) = s_7$	$\delta(s_7, \{e_1^2\}) = s_2$
$\delta(s_8, \{e_1^9\}) = s_9$	$\delta(s_9, \{e_1^{10}\}) = s_{10}$	$\delta(s_{10}, \{e_1^{11}\}) = s_{11}$
$\delta(s_{11}, \{e_1^{12}\}) = s_{12}$	$\delta(s_{12}, \{e_1^4\}) = s_4$	$\delta(s_{13}, \{e_1^{14}\}) = s_{14}$

Based on the definitions of our state model in Section 3.3, we have $S_M = (Q, \delta, s_I, s_k, M)$, where $Q = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}\}$, the initial state is s_1, the final state is s_{14}, and the transition matrix M is computed as follows:

$$\begin{array}{c|cccccccccccccc|}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} & s_{14} \\
s_1 & 0 & .999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_2 & 0 & 0 & .1998 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .8 & 0 \\
s_3 & 0 & 0 & 0 & .998 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_4 & 0 & 0 & 0 & 0 & .8 & 0 & 0 & .1998 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_5 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_6 & 0 & 0 & 0 & 0 & 0 & 0 & .997 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_7 & 0 & .999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .998 & 0 & 0 & 0 & 0 & 0 \\
s_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .8 & 0 & 0 & 0 & 0 \\
s_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 \\
s_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .997 & 0 & 0 \\
s_{12} & 0 & 0 & 0 & .999 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
s_{13} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\
s_{14} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

$$M(1,2) = R_1L_{1,2} = 0.999 \times 1.0 = 0.999$$

$$M(2,3) = R_2L_{2,3}P_{2,3} = 1.0 \times 0.999 \times 0.2 = 0.1998$$

$M(2,13) = R_2L_{2,6}P_{2,6} = 1.0 \times 1.0 \times 0.8 = 0.8$

$M(3,4) = R_3L_{3,2} = 0.998 \times 1.0 = 0.998$

$M(4,5) = R_2L_{2,6}P_{2,6} = 1.0 \times 1.0 \times 0.8 = 0.8$

$M(4,8) = R_2L_{2,3}P_{2,3} = 1.0 \times 0.999 \times 0.2 = 0.1998$

$M(5,6) = R_6L_{3,4} = 1.0 \times 1.0 = 1.0$

$M(6,7) = R_4L_{4,5} = 0.997 \times 1.0 = 0.997$

$M(7,2) = R_5L_{4,2} = 0.999 \times 1.0 = 0.999$

$M(8,9) = R_3L_{3,2} = 0.998 \times 1.0 = 0.998$

$M(9,10) = R_2L_{2,6}P_{2,6} = 1.0 \times 1.0 \times 0.8 = 0.8$

$M(10,11) = R_6L_{3,4} = 1.0 \times 1.0 = 1.0$

$M(11,12) = R_4L_{4,5} = 0.997 \times 1.0 = 0.997$

$M(12,4) = R_5L_{4,2} = 0.999 \times 1.0 = 0.999$

$M(13,14) = R_6L_{1,5} = 1.0 \times 1.0 = 1.0$

$n = 14$, $|(I\text{-}M)_{n,1}| = -0.67222$, $|I\text{-}M| = 0.682235$, $R_n = 0.999$

$$\text{System reliability } R = (-1)^{n+1} R_n \frac{|(I-M)_{n,1}|}{|I-M|} = 0.98434$$

Chapter 6

Conclusions

In this thesis, we developed an architecture-based software reliability model to compute the reliability of large-scale software, addressing the architectural characteristics and complexity, supporting decision-making on choosing a better-fit design in the early stage of software process, and accommodating frequent component upgrades or updates.

We present a two-phase approach. The first phase utilizes architectural styles to identify system structures in a formal way and resolves the modeling limitations of homogeneous reliability models to address heterogeneous software behaviors. The second phase is to relax the assumption of the Markov-based reliability models, which assumes that the software process follows a Markov process, i.e. independent of execution history. In other words, we take advantage of a homogeneous stochastic Markov model, a white-box approach, to ease the modeling of the interrelationships among components. We encompass the heterogeneity of system structures into our model, and address the modeling of execution history, probabilistic and deterministic software behaviors.

Because of using white-box approach, our architecture-based reliability model can provide relative analysis to facilitate decision-making on alternative architectures and the choosing of software components at an early phase of software process without the need for retesting the whole software system.

6.1 Heterogeneous Software Reliability Modeling and Component Sensitivity Measurement

The theme of our study is the evaluation of software reliability taking into account its architecture, including both homogeneous and heterogeneous system structures. We study software architecture, based on the characteristics of high-level architectural styles to realize system configurations, component composition constraints, and high-level semantics.

To compute system reliability, we develop a state model with the identified architectural styles in a system as inputs to address different system structures or heterogeneous architectures, and then apply the methodology of discrete time Markov-based approaches to compute software reliability. Our state model differs from those of traditional homogeneous Markov-based reliability models in several ways by allowing not only one component to one state mapping, but also multiple components to one state mapping. This enables the modeling of more complex system structures, such as parallel computations and fault tolerant architectures, as well as heterogeneous software architectures.

The above concludes our first phase, which allows us to model software that follows Markov properties with either homogeneous or heterogeneous architectures. In addition, we also conduct component sensitivity analysis by differentiating overall system reliability with individual component reliability. This suggests the component improvement sequence to effectively allocate resources and effort. For software with high complexity, an informal approach can also be utilized to realize the critical components to a system.

Because a software process is most likely not following a Markov process, we develop our second phase approach to remove the fundamental barrier of traditional Markov-based reliability models to broaden the application domains.

6.2 Deterministic Software Behaviors and Execution History Modeling

Our modeling of software behaviors and execution history relaxes the Markov process assumption of traditional Markov-based reliability models. The assumption implies that the next component to be executed depends probabilistically on the present component only and is independent of execution history. Because many types of software do not satisfy the Markov process assumption, the applicability of the Markov-based reliability models is thus very limited.

In our second phase approach, the modeling challenges are classifies into three problem domains: deterministic software behaviors, non-terminated processes, and an infinite number of hybrid processes. The deterministic software behaviors of a process are definite as to the next step or next few steps of its execution. A non-terminating process cannot terminate the execution because of an infinite number of component executions. For the third problem, an infinite number of hybrid processes has an unlimited number of execution paths, but each process is involved with execution history and will eventually terminate.

To model deterministic software behaviors, our state model further allows a component to locate in multiple states to address execution history. The creation of a new state is based on the derivation of our grammar production rules, in which a derived terminal represents a state, and a list of terminals represents an execution path. Following the execution paths, the next component to be executed will depend deterministically on the present component and its ancestors.

For a non-terminating process, we treat it as an unreliable process because it yields no expected outcome. In our model, we identify the non-terminals in the grammar that cannot derive terminal strings. These non-terminals are replaced with new productions rules, which derive an *err* terminal to indicate the failure of a software process. Therefore, the model prevents the loop of infinite derivations without yielding any results and at the same time saves the computations of reliability measurement.

For the modeling of an infinite number of hybrid processes, a state model ideally requires an infinite number of states, which limits the usage of finite states discrete time Markov models. We resolve the infinite number of state expansion by utilizing the structure of binomial trees to construct a recurrent loop, in order to exactly model all possible execution paths without over- or under-estimating. Each node of the binomial tree is a set of basic clusters that addresses the execution history. Such a recurrent loop structure reduces the needs of an infinite number of states to a finite number of states. Therefore, a state model can be constructed to compute software reliability.

All in all, combining the first phase with the second phase, our architecture-based reliability model not only addresses heterogeneous system structures, but also significantly broadens the application domains. With its white-box characteristics, relative software quality analyses can be conducted to achieve early prediction and decision-making, which support effective resource allocations and reduce the chance of future software failures. In addition, our architecture-based reliability model is expected to be applicable to the reliability measurement for both software and hardware domains.

Appendix A

Discrete Time Markov Models

In this section, we present the properties of discrete-time Markov models that serve as the foundations of our architecture-based software reliability model. Our model utilizes discrete-time Markov chains to compute system reliability. Because of the white-box approach of discrete time Markov models, it allows the modeling of the interrelationships to a level between two states. A discrete-time Markov model consists of

1. A finite set of n states,
2. A non-negative $n \times n$ stochastic matrix $T = (P_{ij})$, where P_{ij} is the transition probability that the system will move to state s_j, given only that the system is in state s_i, where $P_{ij} \geq 0$, and $\sum_{j=1}^{n} P_{ij} = 1$, $1 \leq i, j \leq n$.
3. A vector $\pi^0 = (\pi_1^0, \ldots, \pi_n^0)$ where π^0 denotes the probability that the system is initially in state s_i, $i = 1, \ldots, n$.

In addition, for a stochastic matrix there are two important lemmas that support our architecture-based reliability model. The detailed proofs are available in [9].

Lemma 1: Let T be a stochastic matrix, which is standardized as shown below, then $\rho(M) < 1$.

$$T = \begin{bmatrix} D_1 & 0 & \cdots & 0 & 0 \\ 0 & D_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & D_r & 0 \\ B_1 & B_2 & \cdots & B_r & M \end{bmatrix} \quad \text{.......... standardized,}$$

where D_i is an $n_i \times n_i$ irreducible state transition matrix associated with its ergodic equivalence class; M is a $k \times k$ square matrix and all the states corresponding to M are transient.

Lemma 2: If M is an $n \times n$ matrix with $\rho(M) < 1$, then $(I\text{-}M)^{-1}$ is nonsingular and $(I–M)^{-1}$ $= \lim_{k \to \infty} \sum_{i=0}^{k} M^i = \sum_{i=0}^{\infty} M^i$, where I is an identity matrix with the same size as M.

When $(I\text{-}M)^{-1}$ is nonsingular, the determinant of I minus M, denoted as $|I\text{-}M|$, is not equal to 0. Therefore, based on these two lemmas, a standardized stochastic matrix T has $|I\text{-}M| \neq 0$.

The following lists the definitions and theorems in [9] that are associated with the above two lemmas:

Definition 1: A state s_i is called *transient* if $s_i \longrightarrow s_j$ for some s_j but $s_j \not\longrightarrow s_i$, that is, s_i has access to some s_j that does not have access to s_i. Otherwise, the state s_i is called *ergodic*. Thus s_i is ergodic if and only if $s_i \longrightarrow s_j$ implies $s_j \longrightarrow s_i$.

Definition 2: The *classes* of a Markov chain are the equivalence classes induced by the communication relation on the set of states. A class α has access to a class β if $s_i \rightarrow s_j$ for some $s_i \in \alpha$ and $s_j \in \beta$. A class is called *final* if it has access to no other class. An *ergodic equivalence class* is a class that contains all ergodic states and the class is final.

Theorem 1: Let $a_{ij}^{(q)}$ denote the (i,j) element of A^q. A nonnegative matrix A is *irreducible* if and only if for every (i,j) there exists a natural number q such that $a_{ij}^{(q)}>0$.

Appendix B

A Homogeneous Markov-Based Reliability Model

In [14], Cheung proposed a reliability model based on discrete-time Markov chains to model homogeneous software, in which common system structures such as branching and module-to-module transitions were modeled. A system with k components is modeled with k mapping states, where state s_i has component c_i activated, $1 \le i \le k$. For generality, a system is considered to have only a single initial state s_1 and a single final state s_k with $\pi_1^0 = 1$. Let R_i be the reliability of component i, and P_{ij} be the transition probability from component c_i to component c_j. A non-negative stochastic matrix T is standardized as follows:

$$T = \begin{array}{c|ccc} & S & F & s_1 \cdots s_k \\ \hline S & 1 & 0 & 0 \\ F & 0 & 1 & 0 \\ s_1 \cdots s_k & B_1 & B_2 & M \end{array}, \quad B_1 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ R_k \end{bmatrix}, \quad B_2 = \begin{bmatrix} 1 - R_1 \\ \vdots \\ 1 - R_{k-1} \\ 1 - R_k \end{bmatrix}$$

$D_1 = [1]$ and $D_2 = [1]$ are irreducible matrices, because any state in the irreducible matrix does not transit to any other state outside the matrix. M is a transient matrix whose transition process will eventually reach either state S or state F. States S and F are two absorbing states, representing the successful output state and failure state, respectively.

Software operates successfully when the system reaches S; otherwise, it fails when the system reaches F. Once the system reaches an absorbing state, the system will not move on to any other states but itself. Therefore, the transition probability from S to S is equal to 1, so is from F to F. In this model, only the final state s_k can reach the successful state S with probability R_k. If component c_i is not always reliable, the probability from state s_i to state F is equal to $1\text{-}R_i$. The entries of a $k \times k$ transition matrix M are calculated as follows, and $M(i, j)$ is the entry value of ith row and jth column.

$$\begin{cases} M(i,j) = R_i P_{ij}, \text{ state } s_i \text{ reaches state } s_j \text{ and } i \neq k \\ M(i,j) = 0, \text{ otherwise} \end{cases}, \text{for } 1 \leq i, j \leq k \text{ .... \{B1\}}$$

The system reliability is computed as $R = Q(1, k)R_k$, which is the probability of considering all possible transitions from the initial state s_1 to the final state s_k times the component reliability R_k of component c_k in s_k. Here,

$$Q(1,k) = I(1,k) + M(1,k) + M^2(1,k) + M^3(1,k) + \ldots\ldots = \sum_{k=0}^{\infty} M^k(1,k) = (-1)^{k+1}\frac{|(I-M)_{k,1}|}{|I-M|}$$

$$\text{System reliability } R = Q(1,k)R_k = (-1)^{k+1}\frac{|(I-M)_{k,1}|}{|I-M|}R_k \text{ .... \{B2\}}$$

I is a $k \times k$ identity matrix and $|I - M|$ is the determinant of matrix $(I - M)$. $|(I - M)_{k,1}|$ is the determinant of the minor matrix, excluding the last row and the first column of the matrix $(I - M)$. Since the constructed stochastic matrix T is standardized, $|I - M| \neq 0$.

Cheung's model considers homogeneous component transitions, branching, and looping. However, this approach is insufficient to model complex structures and heterogeneous architectures, because some structures can have multiple activities taking place simultaneously or require some specific actions based on the running situations. For example, a parallel architecture has multiple components running concurrently to improve performance, a fault tolerant system has backup components compensating the failure of the others to improve reliability, or a call-and-return structure that has caller components invoke callee components many times to eliminate code redundancy. Furthermore, this homogeneous model limits a component to only one state, and assumes that a software process follows a Markov process. Thus, it hinders the modeling of execution history, and most deterministic software behaviors.

Appendix C

An Example of Traditional Markov-Based Software Reliability Modeling

In this section, a simple system is used to demonstrate the utilization of traditional Markov-based reliability models to compute reliability of homogeneous software. This sample system consists of three components *A*, *B*, and *C*, as shown in Figure 21. Basically, this software operates failure free when component *A* eventually transits to component *C*, and then component *C* produces a correct outcome. Assume that components are independent of each other and their component reliabilities R_A, R_B, and R_C are 0.9, 0.95 and 1.0, respectively. Furthermore, we know that component *A* has a chance of 0.6 of transiting to component *B*, while there is also a chance of 0.4 of transiting to component *C*. We will first show the computation of software reliability based on probability theory, and then model the system into a Markov model and yield the same reliability.

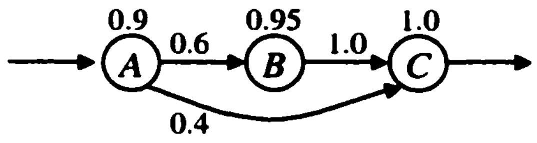


Figure 21: A sample system

C.1 Using Probability Theory

Given this software system in Figure 21, there are two possible execution paths from *A* to *C*. One is from *A* to *B* and then from *B* to *C*, and the other is from *A* directly to *C*. The first execution path requires components *A*, *B*, and *C*, all being reliable, which can be computed as $R_AR_BR_C = 0.9 \times 0.95 \times 1.0 = 0.855$. As we know, the chance of going through the first execution path is 0.6. Therefore, the reliability of the first execution path is equal to 0.6×0.855. Similarly, the second execution path requires components *A*, and *C*, both being reliable, which can be computed as $R_AR_C = 0.9 \times 1.0 = 0.9$. The chance of going through the second execution path is 0.4. The reliability of the second execution path is equal to 0.4×0.9. Software reliability is the sum of the reliabilities of all the execution paths. Therefore, system reliability R is equal to $0.6 \times 0.855 + 0.4 \times 0.9 = 0.873$.

C.2 Using Homogeneous Markov-Based Reliability Model

Based on the same example in Figure 21, our first step is to construct the stochastic matrix *T*, which is standardized as shown below. In addition to states S_A, S_B, and S_C for components *A*, *B*, and *C*, two absorbing states *S* and *F* are added to represent a successful state and a failure state.

$$T = \begin{array}{c|ccccc} & S & F & S_A & S_B & S_C \\ \hline S & 1 & 0 & 0 & 0 & 0 \\ F & 0 & 1 & 0 & 0 & 0 \\ S_A & 0 & 0.1 & 0 & 0.54 & 0.36 \\ S_B & 0 & 0.05 & 0 & 0 & 0.95 \\ S_C & 1 & 0 & 0 & 0 & 0 \end{array} \qquad M = \begin{bmatrix} 0 & 0.54 & 0.36 \\ 0 & 0 & 0.95 \\ 0 & 0 & 0 \end{bmatrix}$$

Here, we take row three of the above stochastic matrix T as an example to illustrate how to fill in the value of an entry. Component A cannot produce the correct outcome itself, so the probability from S_A to S is equal to 0. Component A has 0.9 component reliability, so that there is a 0.1 chance of going to the failure state F. Component A does not transit to itself, so that S_A to S_A is equal to 0. The probability from S_A to S_B is equal to 0.54, which is the possibility that component A transits to component B under the condition that A functions reliably. Similarly, the entry from S_A to S_C is equal to 0.36, which is the possibility that component A transits to component C under the condition that A functions reliably.

$$M^0 = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad M^1 = \begin{bmatrix} 0 & 0.54 & 0.36 \\ 0 & 0 & 0.95 \\ 0 & 0 & 0 \end{bmatrix} \quad M^2 = \begin{bmatrix} 0 & 0 & 0.513 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad M^3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{aligned} & M^3 = M^4 = M^5 = \cdots = M^\infty \\ & \text{Let } Q = M^0 + M^1 + \cdots + M^\infty \\ & R = Q(1,3)R_C = 0.873 \end{aligned} \qquad Q = \begin{bmatrix} 1 & 0.54 & \textcircled{0.873} \\ 0 & 1 & 0.95 \\ 0 & 0 & 1 \end{bmatrix}$$

To compute software reliability, we are interested in the transient matrix M consisting of only those three states S_A, S_B, and S_C. Each entry $M(i,j)$ in the matrix M is the probability from one transient state s_i to another transient state s_j with only one single transition. M^2 is the multiplication of M by M, and each entry in M^2 represents the probability of proceeding through two transitions. Take the 0.513 in the entry of M^2 (1,3) for example, it means that 0.513 is the probability from A to component C by going through two transitions. In this case, it is from A to B, and then B to C, equal to the chance of 0.6 times the reliability 0.855 of the previous first execution path. Similarly, each entry in M^3 is the probability of going through three transitions. By summing up M^0, M^1 to M^∞ together, we obtain a matrix Q. Each entry value in Q depicts the probability of going through all possible numbers of transitions from one state to the other. Take $Q(1,3)$ for example, the entry value 0.873 is the probability of covering all possible execution paths, including the one from A to B to C, and the other one from A directly to C. Software reliability R is then computed as $Q(1,3)R_C$, which is the probability of reaching the exit state S_C, and the reliability of component C that operates failure free.

This sample system is rather simple. However, if state transitions encounter a loop, it becomes infeasible to do all the matrix multiplication and summation. For example, component B can also transit back to component A. Therefore, we can take advantage of Cheung's formula {B2}, derived from linear algebra, to compute software reliability as follows. Note that $|I - M| \neq 0$ because the stochastic matrix T is standardized.

$$\begin{aligned} R &= \sum_{i=0}^{\infty} M^i(1,3)R_C = Q(1,3)R_C = (-1)^4 \frac{|(I-M)_{3,1}|}{|I-M|} R_C \\ &= (-1)^4(0.873)(1)/1 \\ &= 0.873 \end{aligned}$$

$$I-M = \begin{bmatrix} 1 & -0.54 & -0.36 \\ 0 & 1 & -0.95 \\ 0 & 0 & 1 \end{bmatrix} \qquad (I-M)_{3,1} = \begin{bmatrix} -0.54 & -0.36 \\ 1 & -0.95 \end{bmatrix}$$

Bibliography

1. Abd-Allah, A., "Architecture Description Languages State of the Art Presentation," In *Knowledge Summary of the USC-CSE Focused Workshop on Software Architectures*, Center for Software Engineering, University of Southern California, Los Angeles, CA. USA, June 1994.

2. Abd-Allah A., "Composing Heterogeneous Software Architectures", *Technical Report USC-CSE-95-502*, University of Southern California, Los Angeles, CA. USA, 1995.

3. Abowd G., Allen R., and Garlan D., "Formalizing Style to Understand Descriptions of Software Architecture", *Technical Report CMU-CS-95-111*, Carnegie Mellon University, Pittsburgh, PA. USA, January 1995.

4. Allen R. and Garlen D., "Towards Formalized Software Architectures", *Technical Report CMU-CS-92-163*, Carnegie Mellon University, Pittsburgh, PA. USA, July 1992.

5. Allen R. and Garlan D., "A Formal Approach to Software Architectures", in *Proceedings of 3rd IFIP Working Conference on Dependable Computing for Critical Applications*, Mondello, Sicily, Italy, September 14-16, 1992.

6. Allen R. and Garlan D., "Formal Connectors", *Technical Report CMU-CS-94-115*, Carnegie Mellon University, Pittsburgh, PA. USA, 1994.

7. Allen R., Garlan D., "Formalizing Architectural Connection", In *Proceedings of the Sixteenth International Conference on Software Engineering (ICSE)*, Sorrento, Italy, pp 71-80, May 1994.

8. Allen R., Garlan D., "Beyond definition/use: Architectural interconnection", In *Proceedings of the ACM Interface Definition Language Workshop*, 29(8), SIGPLAN Notices, August 1994.

9. Berman A. and Plemmons R. J., *Nonnegative Matrices in the Mathematical Science*, Academic Press Inc., New York, 1979.

10. Binns P., Englehart M., Jackson M., Vestal S., "Domain-Specific Software Architectures for Guidance, Navigation, and Control", *International Journal of Software Engineering and Knowledge Engineering*, 6(2), 1996.

11. Boehm B. W. and Scherlis W. L., "Megaprogramming", In *Proceedings of the DARPA Software Technology Conference*, April 1992.

12. Brooks F. P., Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, 20(4), pp. 10-19, April 1987.

13. Chen M. H., Lyu M. R., Rego V. J., Mathur A. P. and Wong E. W., "Effect of Code Coverage on Software Reliability Measurement", *IEEE Transactions on Reliability*, 50(2), pp. 165-170, June 2001.

14. Cheung R. C., "A User-Oriented Software Reliability Model", *IEEE Transactions On Software Engineering*, 6(2), pp. 118-125, March 1980.

15. Clark B. K., "Domain Specific Software Architecture State of the Art Presentation", In *Proceedings of the USC-CSE Focused Workshop on Software Architectures*, Center for Software Engineering, University of Southern California, Los Angeles, CA. USA, June 1994.

16. Cormen H. T., Leiserson E. C., Rivest L. R., *Introduction to Algorithms*, McGraw-Hill Book Company, Tenth Edition, pp. 401-403, 1993.

17. DeRemer F. and Kron H.H., "Programming-in-the-Large versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, SE-2, pp. 80-86, June 1976.

18. Duane, J. T., "Learning Curve Approach to Reliability Monitoring", *IEEE Transactions on Aerospace*, 2, pp. 563-566, 1964.

19. Dutton G., Sims D., "Patterns in OO Design and Code Could Improve Reuse." *IEEE Software*, 11(3), pp. 101, May 1994.

20. Englehart M., Jackson M., "ControlH: An Algorithm Specification Language and Code Generator", *IEEE Control Systems Magazine*, April 1995.

21. Farr W., "Software Reliability Modeling Survey", In M. R. Lyu editor, *Handbook of Software Reliability Engineering*, McGraw-Hill Publishing Company and IEEE Computer Society Press, New York, pp. 71-117, 1996.

22. Flowers S., *Software Failure: Management Failure, Amazing Stories and Cautionary tales*, John Wiley & Sons Ltd, 1996.

23. Gacek C., "Domain Specific Software Architecture Based Reuse State of the Art Presentation", In *Knowledge Summary of the USC-CSE Focused Workshop on*

Software Reuse, Center for Software Engineering, University of Southern California, Los Angeles, CA. USA, October 1994.

24. Garlan D., "The Role of Formal Reusable Frameworks", In *ACM SIGSOFT Software Engineering Notes: Proceedings of Formal Methods in Software Development*, 15(4), pp. 42-44, September 1990.

25. Garlan D., "What Is Style?", In *Proceedings of First International Workshop on Software Architecture*, Saarbruecken, Germany, April 1995.

26. Garlan D., Allen R., and Ockerbloom J., "Exploiting Style in Architectural Design Environments", In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, 19(5), New Orleans, Louisiana, December 1994.

27. Garlan D. and Shaw M., "An Introduction to Software Architecture", In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

28. Goel, A. L. and Okumoto K., "An Analysis of Recurrent Software Errors in a Real-Time Control System", In *Proceedings of the 1978 annual ACM Conference*, pp. 496-501, 1978.

29. Goel, A. L. and Okumoto K., "Time-Dependent Error-Detection Rate Model for Software Reliability and other Performance Measures", *IEEE Transactions on Reliability*, 28(3), pp. 206-211, 1979.

30. Gokhale S. S., Lyu M. R., and Trivedi K. S., "Reliability Simulation of Component-Based Software Systems", In *Proceedings of Ninth International Symposium on*

Software Reliability Engineering (ISSRE), pp192-201, Paderborn, Germany, November 1998.

31. Gokhale S. S., Wong W. E., Trivedi K. S., and Horgan J. R., "An Analytical Approach to Architecture-based Software Reliability Prediction", In *Proceedings of IEEE International Computer Performance and Dependability Symposium (IPDS)*, Durham, North Carolina, September 1998.

32. Hamlet D., Mason D., and Woit D., "Theory of Software Reliability Based on Components", In *Proceedings of 23rd International Conference on Software Engineering (ICSE)*, Toronto, Ontario, Canada, May 2001.

33. Hoare, C. A. R. "Communicating Sequential Processes", *Communications of the ACM*, 21(8), pp. 666-677, August 1978.

34. Hudson, G. R., "Program Errors as a Birth and Death Process", *Technical Report SP-3011*, System Development Corporation, Santa Monica, CA. USA, 1967.

35. Institute of Electrical and Electronics Engineers, *Software Engineering Standards*, Institute of Electrical and Electronics Engineers, 3rd Edition, New York, NY. USA, 1989.

36. Iannino A., Musa J. D., Okumoto K. and Littlewood B., "Criteria for Software Reliability Model Comparisons", *IEEE Transactions on Software Engineering*, 10(6), pp. 687-691, 1984.

37. Inverardi P. and Wolf A. L., "Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model", *IEEE Transactions on Software Engineering*, 21(4), April 1995.

38. Jelinski Z., and Moranda P. B., *Software Reliability Research*, (W. Freiberger, Editor), Statistical Computer Performance Evaluation, Academic, New York, pp. 465-484, 1972.

39. Johnson A. M., Jr. and Malek, M., "Survey of Software Tools for Evaluating Reliability, Availability, and Serviceability", *ACM Computing Surveys*, 20(4), pp. 227-269, December 1988.

40. Jones A. K., "The Maturing of Software Architecture", In *Proceedings of Software Engineering Symposium*, Software Engineering Institute, Pittsburgh, PA, August 1993.

41. Kremer, W., "Birth-Death and Bug Counting", *IEEE Transactions on Reliability*, 32(1), pp.37-47, 1983.

42. Krishnamurthy S. and Mathur A. P., "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", In *Proceedings of Eighth International Symposium on Software Reliability Engineering (ISSRE)*, pp 146-155, Albuquerque, NM. USA, November 1997.

43. Kyparisis, J. and Singpurwalla, N. D., "Bayesian Inference for the Weibull Process with Applications to Assessing Software Reliability Growth and Predicting Software Failures", *Computer Science and Statistics*, 16th Symposium Interface, Atlanta, Georgia, pp. 57-64, 1984.

44. Laprie J-C., Kanoun K., Béounes C., Kaâniche M., "The KAT (Knowledge-Action-Transformation) Approach to the Modeling and Evaluation of Reliability and

Availability Growth", *IEEE Transactions on Software Engineering*, 17(4), pp. 370-382, April 1991.

45. Laprie J-C. and Kanoun K., "Software Reliability and System Reliability", *Handbook of Software Reliability Engineering*, pp. 27-70, McGraw-Hill, New York, 1996.

46. Li J. J., Micallef J., and Horgan J. R., "Automatic Simulation to Predict Software Architecture Reliability", In *Proceedings of Eighth International Symposium on Software Reliability Engineering (ISSRE)*, pp. 168-179, Albuquerque, NM. USA, November 1997.

47. Littlewood B. and Verrall J. L., "A Bayesian Reliability Growth Model for Computer Software", *Journal Royal Statistical Society-Series C*, 22(3), pp. 332-346, 1973.

48. Littlewood B., "A Reliability Model for Systems with Markov Structure", *Applied Statistics*, 24(2), pp. 172-177, February 1975.

49. Littlewood B., "Software Reliability Model for Modular Program Structure", *IEEE Transactions on Reliability*, 28(3), pp. 241-246, August 1979.

50. Littlewood B., "Stochastic Reliability-Growth: A Model for Fault-Removal in Computer-Programs and Hardware-Design", *IEEE Transactions on Reliability*, 30(4), pp. 313-320, October 1981.

51. Liu G., "A Bayesian Assessing Method of Software Reliability Growth", *Reliability Theory and Applications*, S. Osaki and J. Cao (Editors), World Scientific, pp. 237-244, 1987.

52. Luckham D., Augustin L., Kenney J., Vera J., Bryan D., Mann W., "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, 21(4), pp. 336-355, April 1995.

53. Medvidovic N., Oreizy P., Robbins J. E., and Taylor R. N., "Using Object-Oriented Typing to Support Architectural Design in the C2 Style", In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE4)*, San Francisco, CA. USA, October 1996.

54. Medvidovic N., Oreizy P., and Taylor R. N., "Reuse of Off-the-Shelf Components in C2-Style Architectures". In *Proceedings of 19th International Conference on Software Engineering (ICSE)*, Boston, MA. USA, May 1997.

55. Mettala E. and Graham M. H., "The Domain-Specific Software Architecture Program", *Technical Report CMU/SEI-92-SR-9*, June 1992.

56. Moranda, P. B., "Perditions of Software Reliability During Debugging", In *Proceedings of Annual Reliability and Maintainability Symposium*, Washington, DC. USA, pp. 327-332, 1975.

57. Musa J. D., "A Theory of Software Reliability and its Application", *IEEE Transactions on Software Engineering*, 1(3), pp. 312-327, 1975.

58. Musa J. D., Iannino A., and Okumoto K., *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, New York, 1987.

59. Musa J. D. and Okumoto K., "Software Reliability Models: Concepts, Classification, Comparisons, and Practice", (J. K. Skwirzynski, Editor), *Electronic Systems*

Effectiveness and Life Cycle Costing, NATO ASI Series, F3, Springer-Verlag, Heidelberg, pp. 395-424, 1983.

60. Musa J. D. and Okumoto K., "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement", In *Proceedings of Seventh International Conference on Software Engineering (ICSE)*, Orlando, FL. USA, pp. 230-238, March 1984.

61. Perry D. E. and Wolf A. L., "Foundations for the Study of Software Architecture", In *Proceedings of the ACM Special Interest Group on Software Engineering (SIGSOFT): Software Engineering Notes*, 17(4), pp. 40-52, October, 1992.

62. Ross, S. K., *Stochastic Processes*, John Wiley & Sons, New York, 1983.

63. Royce W., Royce W., "Software Architecture: Integrating Process and Technology", *TRW Space & Defense*, 1991.

64. Schick, G. J. and R. W. Wolverton, "Assessment of Software Reliability", In *Proceedings of Operations Research*, Physica-Verlag, Wurzburg-Wien, pp. 395-422, 1973.

65. Schick, G. J. and R. W. Wolverton, "An Analysis of Competing Software Reliability Models", *IEEE Transactions on Software Engineering*, 4(2), pp. 104-120, 1978.

66. Schneidewind, N. F., "An Approach to Software Reliability Prediction and Quality Control", In *1972 Fall Joint Computer Conference, AFIPS Conference Proceedings*, 41, AFIPS Press, Montvale, NJ. USA, pp. 837-847, 1972.

67. Schneidewind, N. F., "Analysis of Error Processes in Computer Software", In *Proceedings of 1975 International Conference on Reliable Software*, Los Angeles, CA. USA, pp. 337-346, 1975.

68. Shaw M., "Larger Scale Systems Require Higher-Level Abstractions", In *ACM SIGSOFT Software Engineering Notes: Proceedings of Fifth International Workshop on Software Specification and Design*, 14(3), pp. 143-146, Pittsburgh, PA. USA, May 1989.

69. Shaw M., "Heterogeneous Design Idioms for Software Architecture." In *Proceedings of Sixth International Workshop on Software Specification and Design*, pp. 158-165, Como, Italy, October 1991.

70. Shaw M., "Software Architectures for Shared Information Systems." *Technical Report CMU-CS-93-126*, Carnegie Mellon University, Pittsburgh, PA. USA, March 1993.

71. Shaw M. and Garlan D., "Characteristics of Higher-level Languages for Software Architecture", *Technical Report CMU-CS-94-210*, Carnegie Mellon University, Pittsburgh, PA. USA, December 1994.

72. Shaw M., DeLine R., Klein D., Ross T., Young D., Zelesnik G., "Abstractions for Software Architecture and Tools to Support Them." *IEEE Transactions on Software Engineering*, 21(4), pp. 314-335, April 1995.

73. Shooman, M. L., "Probabilistic Models for Software Reliability Prediction", (W. Freidberger, Editor), *Statistical Computer Performance Evaluation*, Academic, New York, pp. 485-502, 1972.

74. Spivey J. M., "An Introduction to Z and Formal Specification", *Software Engineering Journal*, 4(1), pp. 40-50, January 1989.

75. Spivey J. M., *The Z Notation: A Reference Manual*, Englewood Cliffs, New Jersey: Prentice-Hall, 1989.

76. Taylor R. N., Tracz W., Coglianese L., "Software Development Using Domain-Specific Software Architectures", *Technical Report ADAGE-UCI-94-01C*, University of California at Irvine, Irvine, CA. USA, 1994.

77. Taylor R. Medvidovic N., N., Anderson K. M., Whitehead Jr. E. J., Robbins J. E., Nies K. A., Oreizy P., and Dubrow D. L., "A Component and Message-based Architectural Style for GUI Software", *IEEE Transactions on Software Engineering*, 22(6), pp. 390-406, June 1996.

78. Terry A., Papanagopoulos G., Devito M., Coleman N., Erman L., "An Annotated Repository Schema", V.3 Working Draft, 1993.

79. Terry A., Hayes-Roth, F., Erman, L., Coleman, N., and Hayes-Roth, B., "Overview of Teknowledge's Domain-Specific Software Architecture Program". In *ACM SIGSOFT Software Engineering Notes*, 19(4), pp 68-76, October 1994.

80. Tracz W., "LILEANNA: A Parameterized Programming Language", Reprinted from In *Proceedings of the 2nd International Workshop on Software Reuse*, Lucca, Italy, pp. 66-78, March 1993.

81. Tracz W., "DSSA (Domain-Specific Software Architecture) Pedagogical Example", *Loral Contributed Documentation and Technical Report, ADAGE-LOR-94-13*, April 1995.

82. Tracz W., "DSSA frequently asked questions (FAQ)", In *ACM SIGSOFT Software Engineering Notes*, 19(2), pp. 52-56, June 1995.

83. Vestal S., "A Cursory Overview and Comparison of Four Architecture Description Languages", *Technical Report*, Honeywell Technology Center, Minneapolis, MN. USA, February 1993.

84. Wagoner, W. L., "The Final Report on a Software Reliability Measurement Study", Technical Report TOR-0074(4112)-1, Aerospace Corporation, El Segundo, CA. USA, August 1973.

85. Whittaker J. A., "Markov Analysis of Software Specifications", *ACM Transactions on Software Engineering and Methodology*, 2(1), pp 93-106, January 1993.

86. Yamada, S., Ohba M. and Osaki S., "S-Shaped Reliability Growth Modeling for Software Error Detection", *IEEE Transactions on Reliability*, 32(5), pp. 475-478, December 1983.

87. Yamada, S. and Osaki S., "Software Reliability Growth Modeling: Models and Assumptions", *IEEE Transactions on Software Engineering*, 11(12), pp. 1431-1437, December 1985.