Designing Reliability Model based on Client-Server Architectural Style for Component Based System

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

In

Software Engineering

Submitted By
Arpita Sharma
(Roll No. 800931005)

Under the supervision of:
Ms. Shivani Goel
(Assistant Professor)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2011

∞

This Work is dedicated to My Parents

∞

Certificate

I hereby certify that the work which is being presented in the thesis entitled, ***"Designing Reliability Model based on Client-Server Architectural Style for Component Based System"***, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Ms. Shivani Goel* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

Arpita Sharma

(Arpita Sharma)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Shivani Goel.

(Ms. Shivani Goel)
Assistant Professor,
Computer Science and Engineering Department
Thapar University
Patiala

Countersigned by

(Dr. Maninder Singh)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

Many people have shared their time and expertise to help me accomplish my goal. First, I would like to sincerely thank my guide, Ms. Shivani Goel, Assistant Professor, CSED, Thapar University, Patiala for her constant support and guidance. Her instant responses to my countless inquiries have been invaluable and motivational. It was a great opportunity to work under her supervision.

Then I would like to thank Dr. Maninder Singh, Head of the Department, CSED, Thapar University, Patiala for providing all the facilities and environment. I would also like to thank all my Teachers for their stimulating discussion and invaluable suggestions during the period of my work.

I would also like to thank my parents for their teaching to always follow the dreams. With their blessings, love and support every work in the world seems possible.

Finally, I wish to thank my friends Ravneet Grewal, Ravneet Chawla, Vaneet, Aman, Aarti, Ipneet, Aradhna and Sonam for their immense love and encouragement without which it would not have been possible to complete this work.

Last but not the least I would like to thank God who have always been with me in my good and bad times.

Arpita Sharma
(800931005)

Abstract

Software development has come a long way from traditional software development, which is characterized by the structured programming paradigm introduced in the late 60's and early 70's to contemporary development practices, which characterize a software application as interacting, independent components.

Modeling and estimating software reliability during testing is useful in quantifying the quality of the software systems. However, such measurements applied late in the development process leave too little to be done to improve the quality of the software system in a cost-effective way. Reliability, an important attribute, is defined as the probability that the system performs its intended functionality under specified design limits. We argue that reliability models must be built to predict the system reliability at the initial phases of development process, and specifically when the implementation artifacts are unavailable.

A number of architecture-based software reliability models have been proposed by researchers which are used for predicting reliability. One of the categories of reliability models is state based model for terminating applications which uses control flow graph to represent the architecture of the system. In this thesis we extend an analytical model for estimating architecture-based software reliability. We extend the model by adding the client server architectural style in it.

Table of Contents

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

This chapter introduces a description of the work presented in thesis. It gives a brief introduction of Component Based Software Development (CBSD), concept of software reliability, software architecture, style based reliability model and the organization of thesis.

1.1 Component Based Software Engineering

"A component is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture [1]*."*

Component-based software engineering (CBSE) is a sub-discipline of software engineering. CBSE is a process that emphasizes design and construction of a computer based systems using reusable software components.

Clements describes CBSE as, "It is changing the way large software systems are developed. CBSE embodies the 'buy, don't build' philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, CBSE shifts the emphasis from programming software to composing software systems. There is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality" [1].

Component based software development is understood to require reusable components that interact with each other and fit into system architectures. CBSE is primarily concerned with three functions [2]:

- Developing software from pre-produced parts
- The ability to reuse those parts in other applications
- Easily maintaining and customizing those parts to produce new functions and features

1.2 Software Reliability

Software Reliability is an important attribute of software quality. Informally denotes a product's trustworthiness or dependability. According to ANSI, Software Reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection [27]. Software reliability has been defined as the probability that no failure occurs in a specified environment during a specified exposure period. For some programs the appropriate time unit of exposure period is the calendar or CPU time, and for some other programs the appropriate time unit of exposure period is an application run corresponding to a selection of an input case from the input case domain (ICD) of the programs [3].

The high complexity of software is the major contributing factor of software reliability problems. Measurement in software is still in its infancy. No good quantitative methods have been developed to represent software reliability without excessive limitations. Various approaches can be used to improve the reliability of software, however, it is hard to balance development time and budget with software reliability. Software reliability techniques are aimed at reducing or eliminating failures in software systems.

For measuring and predicting system reliability, following basic notions are used: mean time to failure (MTTF) defines the average time to the next failure; mean time to repair (MTTR) is the average time it takes to diagnose and correct a fault, including any reassembly and restart times; mean time between failures (MTBF) is simply defined as MTBF = MTTF + MTTR. The failure rate is the number of failures per unit time, it is reciprocal to MTBF. Another important concept closely related to reliability is availability. This is defined as the probability of a system being available when needed. Availability, or more specifically, instantaneous availability, is typically defined as the fraction of time during which a component or system is functioning acceptably, i.e., the uptime over the total service time

$$A = MTTF / MTBF = MTTF / (MTTF + MTTR)$$

In particular hardware systems, MTTF and MTTR are measured. For many systems, the failure rate and thus MTBF is constant-assuming that system changes can be ignored. The MTBF is then proportional to the length of time considered and is

equated to the reliability. Moreover, repair times are often not meaningful for software, or, repairs may introduce faults. Therefore failure rate is more commonly used as a basis for software reliability measurement. Since a particular software component is not running all the time, we measure its reliability relative to execution time or the number of calls. This fits well with our notion of protocols of behavior specified by finite state machines (FSMs) or Petri nets. The execution of a protocol successively ‘‘fires’’ transitions and failure rates are relative to the length of firing sequences [14].

Reliability must be built into a software system at the initial level of the development process, and as an innate aspect of system design. This requires developing and/or adapting reliability models to predict and measure the reliability of a software system early on. After all, “you can’t control what you can’t measure”

1.3 Software Architecture and Styles

The software architecture of a system consists of software components, their external properties, and their relationships with one another. As software architecture is the foundation of the final software product, the design and management of software architecture is becoming the dominant factor in software reliability engineering research. Well designed software architecture not only provides a strong, reliable basis for the subsequent software development and maintenance phases, but also offers various options for fault avoidance and fault tolerance in achieving high reliability [7].

An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. One can think of architecture styles and patterns as sets of principles that shape an application. Garlan and Shaw define architectural style as, “Family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary 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. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say,

having to do with execution semantics—might also be part of the style definition [28]."

1.4 Style Based Reliability Model

It is used to model the reliability of an architecture style based on its characteristics. A model that computes the reliability of software composed of heterogeneous architectural styles.

For a system with a complex architectural environment, the reliability model needs to be further refined to take the characteristics of different architectural styles into account. The architecture-based reliability model to compute the reliability of heterogeneous software systems may be comprised of various architectural styles like pipe and filter, batch-sequential, call and return etc. [20]. Here the focus is on client server based architectural style.

In the client server style, the client component may request some services which are provided by the server components. A component can act as a server to one component but may act as a client to other component. Therefore, the components may act as client or server at the same time.

1.5 Organization of Thesis

The chapters in this thesis are organized as follows:

Chapter 2 -- This chapter describes in detail the literature survey. It covers the details of software component reliability, its importance, types of reliability models, software architecture and style based reliability modeling.

Chapter 3 – This chapter describes the problem statement of the thesis and objectives.

Chapter 4 – This chapter describes the solution of the problem i.e. the proposed reliability model.

Chapter 5 – This describes the implementation of the proposed reliability model and its validation.

Chapter 6 – This describes the conclusion and future work that can be carried based on the work presented in this thesis.

CHAPTER 2
LITERATURE REVIEW

This chapter describes in detail the literature survey. It covers the details of software component reliability, its importance, types of reliability models, software architecture and style based reliability modeling.

2.1 Software Reliability in Component Based System

The reliability measurement of component based system comprises of two main activities. First, the reliability measurement of individual component and secondly, the reliability measurement of the whole system based on the component context model, as shown in Figure 2.1.

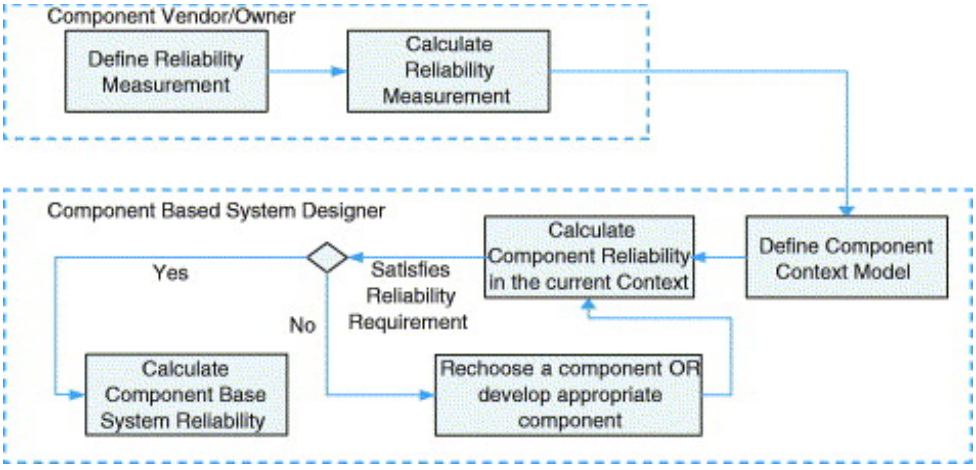

Figure 2.1: Component based system reliability process [4]

In Component Based System (CBS), every component accomplishes a task independently, whose frame inside is transparent to users. All components cooperate with each other by interfaces to achieve the whole system's function. These characteristics bring many difficulties in calculating the reliability of CBS [5].
They are detailed as follows:

- The reliability of every component is in doubt. Every component may be developed by different teams, tools and persons. Almost all CBS is made up of

COTS, although every COTS has particular notes and data of reliability, we can only handle the interfaces between components.

- The adaptive ability of every component is different. When a component is embedded in a software system, its reliability will be different because of different running environment.

2.1.1 Importance of Software Reliability

Software reliability is cited by most users of software products as the most important feature of the software products they use. Hence it is crucial that software reliability engineering techniques should play a central role in the planning and control of software development projects. In particular, it is important to document the times and nature of bug occurrences, and their correction times, throughout the design and integration phases as well as during the testing phase [27]. With such data it is possible to estimate suitable reliability models which can be used to estimate the time at which the software product will have reached a target level of reliability, or to devise methods to decrease that time. Other importance things that can be achieved through software reliability are as follows [3, 27]:

- Determine whether software can be released
- Predict resources required to bring software to required reliability
- Determine impact of insufficient resources on operation reliability
- Prioritize testing or inspection of modules having highest estimated fault content
- Develop fault avoidance technique
- Minimize no of faults inserted
- Prevent insertion of specific types of faults
- Estimate or predict the quality of the software

2.2 Reliability Models

Software reliability growth models are a statistical interpolation of defect detection data by mathematical functions. The functions are used to predict future failure rates or the number of residual defects in the code. Software modeling techniques can be divided into two subcategories [7]:

- prediction modeling
- estimation modeling

Both kinds of modeling techniques are based on observing and accumulating failure data and analyzing with statistical inference.

There are essentially two types of software reliability models - those that attempt to predict software reliability from design parameters and those that attempt to predict software reliability from test data. The first type of models are usually called "defect density" models and use code characteristics such as lines of code, nesting of loops, external references, input/outputs, and so forth to estimate the number of defects in the software. The second type of model is usually called "software reliability growth" models. These models attempt to statistically correlate defect detection data with known functions such as an exponential function. If the correlation is good, the known function can be used to predict future behavior. Software reliability growth models are the focus of this report [9].

Software reliability models are classified according to the development phases of software life-cycle. Since fault corrections are necessary in the testing and debugging phase of the lifecycle, reliability growth models which take fault corrections into account are mainly used in this phase. The models without dealing with fault corrections, say input domain models, can only be applied in this phase by treating the program after each correction as a new program.

Goel [35] categorized software reliability models according to the nature of failure process into four types: times between- failures models, failure-count models, fault seeding models, and input domain based models. The times-between-failures models and failure-count models are usually applied in the iterations of the testing and debugging phase in the software development process based on the fault corrections and reliability history. It is normally assumed that the reliability of software increases with the removals of faults in the software and thus these types of models are also called reliability growth models [29]. On the other hands, fault seeding models and input domain based models do not consider the fault correcting activities. They are

usually used at the end of the development cycle to assess the final reliability of software.

All existing software reliability models are developed for the software products that are statically constructed, normally by a company or institution that has the full control of the development process.

2.2.1 Types of Reliability Models

- *Early prediction models:* This type of models uses characteristics of the software development process from requirements to test and extrapolate this information to predict the behavior of software during operation [10].

- *Software reliability growth models (SRGM):* This type of models captures failure behavior of software during testing and extrapolates it to determine its behavior during operation. Hence this category of models uses failure data information and trends observed in the failure data to derive reliability predictions [9]. The SRGMs are further classified as Concave models and S-shaped models [9].

- *Input domain based models:* These models use properties of the input domain of the software to derive a correctness probability estimate from test cases that executed properly [11].

- *Architecture based models:* This type of models puts emphasis on the architecture of the software and derives reliability estimates by combining estimates obtained for the different modules of the software [12]. The architecture based software reliability models are further classified into State based models; Path based models and Additive models.

- *Hybrid black box models:* These models combine the features of input domain based models and software reliability growth models [29].

- *Hybrid white box models:* These models use selected features from both white box models and black box models. However, since these models consider the architecture of the system for reliability prediction, therefore these models are considered in hybrid white box models [29].

2.2.2 Need of Software Reliability Models

Here the only concern is with reliability analysis model for component-based software. These are as follows [6]:

- Analyze the reliability of a component-based application even when the source code of its components is not available.
- Develop a probabilistic technique for reliability analysis that is applicable at the design-level, before the actual development and integration phases. Many reliability analysis techniques use test cases and fault injection to study the reliability of component-based systems.
- Study the sensitivity of the application reliability to reliabilities of components and interfaces. This could guide the process of identifying critical components and interfaces and analyze the effect of replacing components with new ones with similar interfaces but with improved estimated reliability.
- Incorporate the effects of interface reliabilities in a probabilistic model for reliability analysis.
- Develop a technique to analyze the reliability of applications built from reusable software components. The emerging field of component-based software engineering exacerbates the need for estimating component reliabilities and analyzing the reliability of component-based applications.

2.3 Architecture Based Reliability Models

The architecture determines the contribution of the reliability of each atomic component to that of the system. Hence the approach is named as architecture-based reliability model. In architecture–based approach, one must model the interaction of all components. Many architecture–based software reliability models have been proposed in the past, mostly by ad hoc methods. Software reliability growth models can be applied to each software component exploiting component's failure data obtained during testing [13]. However, due to the scarcity of failure data it is not always possible to use software reliability growth models.

2.3.1 Need for Architecture Based Reliability Model

System reliability cannot be equated to software component reliability. Component interactions make a system more than the sum of its parts - and make system reliability a very complex design-specific function of external component reliabilities and the probability of rare human failure. This is shown for instance in the well documented Therac-25 failure [30]. With the increasing interoperation and networking of software systems, the increasing capability and speed of communication between components and systems, errors can spread widely before humans can intervene. Fault-tolerance requires a systematic and formal approach to reliability.

But component-based models for reliability, especially compositional ones are lacking. Software reliability is defined as the probability of failure-free operation of a software system for a specified period of time in a specified environment. It is a function of the software faults and its operational profile, i.e., the inputs to and the use of the software. For open systems, reliability is also a function of the reliability of essential required services in the deployment context of a software component. Some other needs of architecture–based software reliability approach include the following [12]:

- System reliability depends on its component reliabilities and their interaction.
- Sensitivity of the application reliability to reliabilities of components and interfaces.
- Process of identifying critical components and interfaces for a given architecture.
- Selecting an architecture that is most appropriate for the system under study.
- Reducing the development cost, maintenance cost and re-engineering cost.
- To avoid implementation level artifacts.

2.3.2 Types of Architecture Based Reliability Model

They are basically divided into three types as shown in Figure 2.2. The models are discussed in detail one by one.

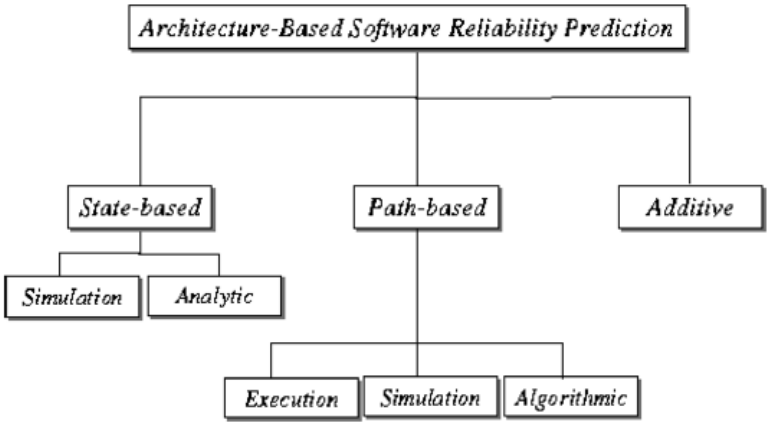

Figure 2.2: Classification of architecture-based software reliability models [16]

2.3.2.1 State-based Models

State based models uses the control flow graph to represent the architecture of the system. It is assumed that the transfer of control between modules has a Markov property which means that given the knowledge of the module in control at any given time, the future behavior of the system is conditionally independent of the past behavior [15]. The architecture of the software has been modeled as a Discrete Time Markov Chain (DTMC), Continuous Time Markov chain (CTMC), or Semi-Markov Process (SMP). These can be further classified into absorbing and irreducible. The former represents applications that operate on demand for which software runs that correspond to terminating execution can be clearly identified. The latter is well suited for continuously operating software applications, such that in real time control systems, where it is either difficult to determine what constitutes a run or there may be very large number of such runs if it is assumed that each cycle consists a run [15].

There are three models for calculating reliability of terminating applications by state based reliability model at architectural level. Cheung Model is one of the earliest models that consider software reliability with respect to the components utilization and their reliabilities. The transfer of control among components is described by an absorbing DTMC with a transition probability matrix $P = [p_{ij}]$, where $p_{ij} = P_r$ {program transits from component *i* to component *j*}. Components fail independently and the reliability of the component *i* is the probability R_i that the component performs its function correctly [17].

Kubat Model includes the information about execution time of each component, thus resulting in an SMP as a model of software architecture. Transitions between components follow a DTMC with initial state probability vector q = [q_{ij}] and transition probability matrix P = [p_{ij}]. The time during a visit in component *i* have the pdf (probability density function) g_i (t). When component *i* is executed, failures occur with constant failure intensity λ_i [18].

Gokhale et al. model is a hierarchical model which differs in the approach taken to estimate the component reliabilities. It considers time dependent failure rates λ_i (t) and the utilization of the components through the cumulative expected time spent in the component per execution $V_i t_i$, where t_i is the expected time spent in a component *i* per visit and V_i is the expected number of times the application visits state i [19].

Assumptions

The three models have different assumptions. For Cheung model it is assumed that the program flow graph of a terminating application has a single entry and a single exit node, and that the transfer of control among modules can be described by an absorbing DTMC [17].

Kubat model considers a software system which has been designed for execution of K different programs or tasks. When a program is called, the modules are executed in certain order, one after another; some modules can be called more than once. So, it is assumed that exchanges of control among the modules can be described by a Markov process, i.e. the probability of calling a given module is a function of the module currently being executed and the calling module only. Also, the failure rate in the module is assumed to be constant and independent of the program type [15].

Gokhale model along with all the above reliability models assume that components fail independently and that a component failure will ultimately lead to system failure [19].

2.3.2.2 Path-based Models

A path–based model computes software reliability considering the possible execution paths of the program. A sequence of components along different paths is obtained

either experimentally by testing or algorithmically. The reliability of each path is computed by multiplying the reliabilities of the components along that path. Then, the system reliability is estimated by averaging path reliabilities over all paths. Path-based models consider software architecture explicitly and assume that components fail independently. However, the method of combining software architecture with failure behavior is not analytical. Path–based models consider different paths that can be taken during software execution. They account for component utilization along each path, as well as among different paths [13].

First, the sequence of components executed along each path is obtained either experimentally by testing or algorithmically and the path reliability is obtained by multiplying the reliabilities of the components along that path. Then, the system reliability is estimated by averaging path reliabilities over all paths.

The comparison chart of different type of path based model is shown in Table 2.1.

Table 2.1: Comparison Chart for Path Based Models

	Shooman Model [22]	Krishnamurthy & Mathur Model [23]	Yocoub, Cukic & Ammar Model [24]
Approach	Path based approach	Experimental approach	Algorithmic approach
Architecture	Assumes the knowledge of the different paths and the frequencies f_i with which path i is run.	Sequences of component among different paths are observed using the component traces collected during testing.	Component Dependency Graph is constructed and the algorithm expands all branches starting from entry node
Failure Behavior	The failure probability of the path i on each run, denoted by q_i, characterize the failure behavior.	Each component is characterized by its reliability R_m.	The failure process considers component reliabilities R_i and transition reliabilities (1 − vij) associated with a node ni and with a transition from node ni to n_j, respectively.

Method of Analysis	The total number of failures nf in N test runs is given by $n_f = \sum_{i=1}^{m} Nf_i q_i$. Where Nf_i is the total number of traversals of path i. The system probability of failure on any test run is given by $q_0 = \lim_{N\to\infty} \frac{n_f}{N} = \sum_{i=1}^{m} f_i q_i$.	The reliability of a path in program P traversed when P is executed on test case tc ∈ TS is given by $R_{tc} = \prod_{\forall m \in M(P,tc)} R_m$. The reliability estimate of a program with respect to a test set TS is $R = \frac{\sum_{\forall tc \in TS} R_{tc}}{\lvert TS \rvert}$.	A tree traversal algorithm based on CDG is used to estimate reliability of the application. Algorithm expands all the branches of CDG. The breath expansions represent logical "OR" paths and depth expansions represent logical "AND" paths.

2.3.2.3 Additive Models

This class of models does not consider explicitly the architecture of the software. Rather, they are focused on estimating the overall application reliability using the component's failure data. They are called additive models since under the assumptions that component's reliability can be modeled by NHPP the system failure intensity can be expressed as the sum of component failure intensities.

2.4 Software Architecture

Software behavior with respect to the manner in which different components interact is defined through the software architecture. Interaction occurs only by transfer of execution control [21]. In the case of sequential software, at each instant, control lies in one and only one of the components. In architecture–based approach, one must model the interaction of all components. In well designed system, interaction among components is limited. During the early phases of software life cycle, each component could be examined to find with which components it cannot interact. If control can flow between two components it can be described by non–zero transition probabilities that may be available by analyzing program structure and using known operational profiles. During the design phase, before actual development and integration phases, the transition probabilities can be estimated by simulation. As new data become available during the integration phase the estimates have to be updated [13].

Software architecture analysis aims at investigating how architecture meets its quality requirements based on the structure and the correlation among the components of the software. It not only facilitates component-based software development, but also provides a means for early quality prediction. Therefore, the quality of component-based software can be predicted by using software architecture analysis methodologies.

Software architecture, which describes the structure of software at an abstract level, consists of a set of components, connectors and configurations [31]. Furthermore, a repeatable pattern that characterizes the configurations of components and connectors of software architectures is considered as an architectural style. Many architectural styles have been identified with new styles continuously emerging. Thus, a practitioner is faced with the challenge of selecting suitable styles, or modeling configurations of selected styles, for designing the architecture to a given software specification. In such a situation, a method or model to predict or evaluate the reliability of a heterogeneous style software system can certainly provide a means through which designers can configure the architecture that best fits their quality demands [32].

At architecture design stage it is possible to select an architectural style that can provide better performance and/or availability than others, if the characteristics of the application environment can be realized. The reliability of the system combines heterogeneous architectural styles including batch-sequential/pipeline, parallel/pipe-filters, call-and-return, and fault tolerance [13]. System reliability is analyzed on the basis of the reliability of components and connectors in these architectural styles. In addition, the operational profile is taken into account by utilizing the transition probabilities from one component to others. Assuming that the reliabilities of components and connectors are independent of the transition probabilities, a Markov model can be utilized to predict the reliability of heterogeneous software architecture, following the transformation from architecture view to state view. Moreover, a system embedded with three architectural styles is utilized to software reliability model [17].

The estimation of transition probabilities is affected by the user's operational profile. Upgrades to software might invalidate any existing estimate of operational profile

because new features can change the ways in which the software is used. Therefore, it will be necessary to revise the architecture that describes component interaction and modify transition probabilities.

2.5 Software Architectural Styles

Software architecture includes a number of architectural patterns, or styles. Some of these representative, broadly-used architectural styles are covered here. To make sense of the differences between styles, a common framework is used to view them. The framework adopted treats an architecture of a specific system as a collection of computational components—or simply components—together with a description of the interactions between these components—the connectors. Graphically speaking, this leads to a view of an abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors [21]. Various software architectural styles are discussed below:

2.5.1 Pipes and Filters

In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed [33]. Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”. Figure 2.3 illustrates this style.

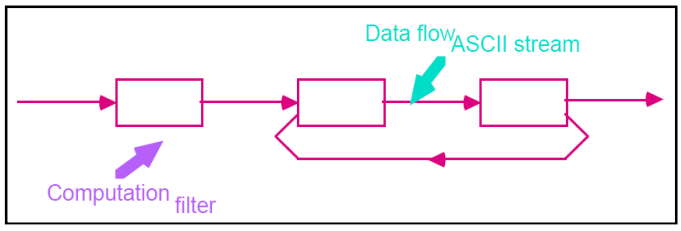

Figure 2.3: Pipes and Filters [8]

2.5.2 Data Abstraction and Object-Oriented Organization

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects—or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource. Objects interact through function and procedure invocations. Two important aspects of this style are (a) that an object is responsible for preserving the integrity of its representation, and (b) that the representation is hidden from other objects [34]. Figure 2.4 illustrates this style.

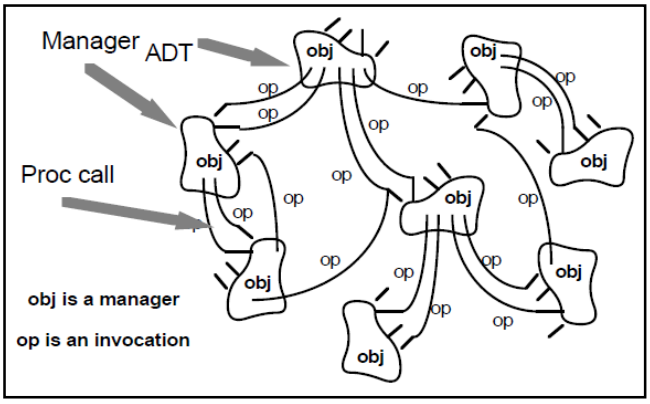

Figure2.4: Abstract Data Types & Objects [34]

2.5.3 Event-based, Implicit Invocation

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. However, there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast [31].

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event [8]. Thus an event announcement “implicitly” causes the invocation of procedures in other modules.

2.5.4 Layered Systems

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers [8]. Figure 2.5 illustrates this style.

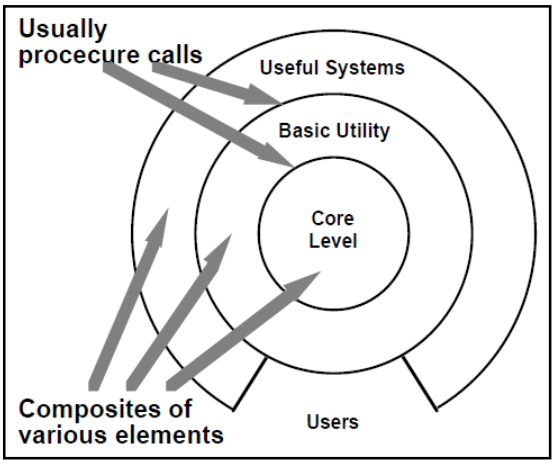

Figure 2.5: Layered System [28]

2.5.5 Repositories

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems [28]. The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard [21]. Figure 2.6 illustrates a simple view of blackboard architecture.

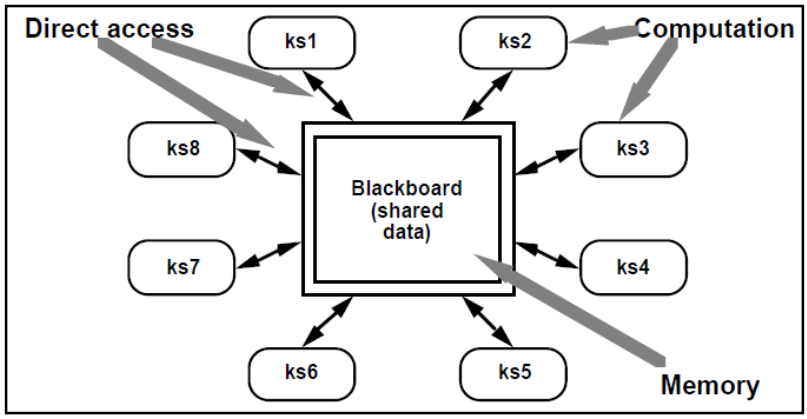

Figure 2.6: The Blackboard [8]

2.5.6 Table Driven Interpreters

An interpreter organization a virtual machine is produced in software. An interpreter includes the pseudo-program being interpreted and the interpretation engine itself. The pseudo-program includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. Thus an interpreter generally has four components: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated.

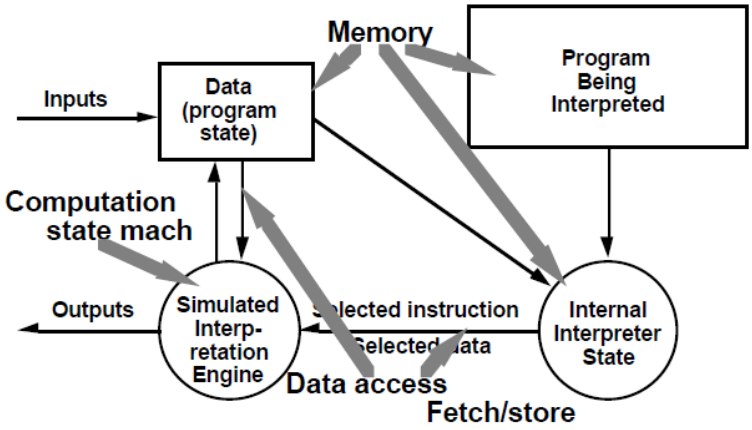

Figure 2.7: Interpreter [8]

2.5.7 Distributed processes

Distributed systems have developed a number of common organizations for multi-process systems. Some can be characterized primarily by their topological features, such as ring and star organizations. Others are better characterized in terms of the kinds of inter-process protocols that are used for communication [21].

2.5.8 Client-Server

One common form of distributed system architecture is a “client-server” organization. In these systems a server represents a process that provides services to other processes (the clients). Usually the server does not know in advance the identities or number of clients that will access it at run time. On the other hand, clients know the identity of a server (or can find it out through some other server) and access it by remote procedure call [21].

2.5.9 Heterogeneous Architectures

Thus far only “pure” architectural styles are studied. While it is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles. There are different ways in which architectural styles can be combined [28].

One way is through hierarchy. A component of a system organized in one architectural style may have an internal structure that is developed a completely different style. For example, in a UNIX pipeline the individual components may be represented internally using virtually any style—including, of course, another pipe and filter, system. What is perhaps more surprising is that connectors, too, can often be hierarchically decomposed [21]. For example, a pipe connector may be implemented internally as a FIFO queue accessed by insert and remove operations [28].

A second way for styles to be combined is to permit a single component to use a mixture of architectural connectors. For example, a component might access a repository through part of its interface, but interact through pipes with other components in a system, and accept control information through another part of its interface [21].

A third way for styles to be combined is to completely elaborate one level of architectural description in a completely different architectural style [28].

2.6 Style Based Reliability Models

Component-based reliability model developed by Cheung [17] is described that takes the reliability of each component and the operational profile into account. In this model, a state diagram which depicts the system behavior is used. A state represents the execution of a single component and the transition probability from one state to another is obtained from the operational profile of a system. The reliability of a software system depends on the execution sequence of states and the reliability of each individual state. Based on Markov chain properties, the transition between states is assumed as a Markov process. It means that components to be executed in next state will depend only on the components of current state and the components of the next state will not have any dependency to the past history of current state. The state diagram is a directed graph in which each node Si represents a state and the transition from state S_i to S_j is represented by a directed edge (S_i, S_j). Let R_i denote the reliability of S_i and P_{ij} be the reaching probability from S_i to S_j. Based on the state diagram, the transition matrix, M, and the value of the entry M(i,j), which can be

$$M = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_i \\ \vdots \\ S_{n-1} \\ S_n \end{array} \begin{array}{c} \begin{array}{ccccccc} S_1 & S_2 & .. & S_i & .. & S_{n-1} & S_n \end{array} \\ \begin{bmatrix} 0 & R_1P_{12} & .. & R_1P_{1i} & .. & R_1P_{1(n-1)} & R_1P_{1n} \\ R_2P_{21} & 0 & .. & R_2P_{2i} & .. & R_2P_{2(n-1)} & R_2P_{2n} \\ \vdots & \vdots & .. & \vdots & .. & \vdots & \vdots \\ R_iP_{i1} & R_iP_{i1} & .. & 0 & .. & R_iP_{i(n-1)} & R_iP_{in} \\ \vdots & \vdots & .. & \vdots & .. & \vdots & \vdots \\ R_{n-1}P_{(n-1)1} & R_{n-1}P_{(n-1)2} & .. & R_{n-1}P_{(n-1)i} & .. & 0 & R_{n-1}P_{(n-1)n} \\ R_nP_{n1} & R_nP_{n2} & .. & R_nP_{ni} & .. & R_nP_{n(n-1)} & 0 \end{bmatrix} \end{array} \quad \ldots.(1)$$

Computed as $R_i \times P_{ij}$, indicates the successful arrival at state S_j from S_i, i.e., the correct execution of the Si and the occurrence of the transition from S_i to S_j.

Below we describe how to calculate the reliability from the transition matrix. Let S = $\{S_1, S_2 \ldots S_n\}$ be the set of states in the state diagram where S_1 is the initial state and S_n is the final state. M^k (i, j) represents the probability of reaching state S_j from S_i through k transitions. Therefore, the reliability R beginning from S_i to S_j with total k transitions is represented as

$$R = M^k(i, j) \times R_j$$

From initial state S_1 to final state S_n, the number of transitions k may vary from 0 to infinity, where 0 means that the initial state is also the final state and infinity means that a cyclic loop may occur indefinitely among the states. Therefore, it is necessary to consider every possible outcome of state transitions. Let T be a matrix such that

$$T = I + M + M^2 + M^3 + \ldots\ldots\ldots = \sum_{k=0}^{\infty} M^k$$
$$\approx \frac{I}{I - M} = (I - M)^{-1}$$

Where I is the identity matrix of size n×n . The overall system reliability can be computed as follows:

$$R = T(1, n) \times Rn$$

With

$$T(1, n) = (-1)^{n+1} \frac{|E|}{|I - M|}$$

Where I - M is the determinant of matrix (I - M) and E is the determinant of the remaining matrix excluding the nth row and first column of the matrix (I - M).

Cheung's model is based on one single initial state and one single final state, whereas a large system may have a set of initial states I = { S_{i1} , S_{i2} , ….., S_{im} } and a set of final states F = { S_{f1} , S_{f2} , ….., S_{fn} }. In such a situation, we revised the problem of multiple initial and final states to one initial state and one final state problem by introducing a super-initial and a super-final state to the state diagram. Figure 2.7 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. We add a super-initial state S_I and a directed edge (S_I , S_{ij}) with its transition probability P_{Ij} , observed from the operational profile, for each j = 1, 2, …., m. Similarly, we create a super-final state S F and a directed edge (S_{fj} , S F) with transition probability 1 for each j = 1, 2, …. , n. The reliabilities for both states S I and S F are assigned 1.

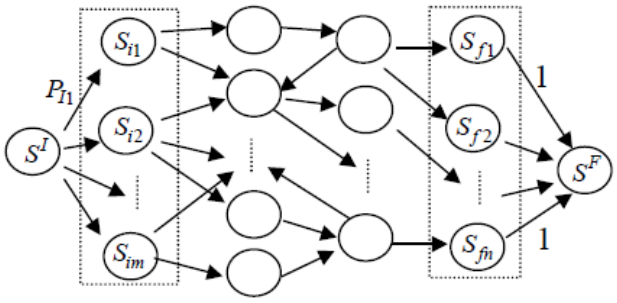

Figure 2.8: One super-initial and one super-final State Diagram [25]

2.6.1 Batch-Sequential/Pipeline Style

Both batch-sequential and pipeline styles are running in a sequential order. They share the same architecture view and state view. Although in the batch-sequential style, outputs of a component are produced only after all its inputs are fully processed, whereas in pipeline style, output may be produced before inputs are fully consumed [26]. These styles can be modeled as shown in Figure 2.8(a), where *C*1 , *C*2 , … ,*Ck* are the components of the architecture and a component such as *C*2 can only go to either one of its branching subsequent components. The transformation from architecture view to state view is one-to-one mapping, shown in Figure 2.8(b), where *S*1, *S*2…, *Sk* are the states of the Markov chain.

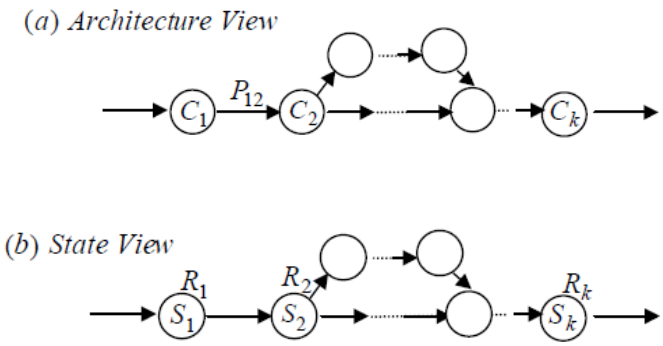

Figure 2.9: Batch-sequential/pipeline style [26]

Assuming that the architecture is composed of k components, in these styles there will be k states in the Markov chain. The transition matrix M can be obtained as follows:

$$\begin{cases} M(i, j) = R_i P_{ij}, & S_i \text{ can reach } S_j \\ M(i, j) = 0, & S_i \text{ cannot reach } S_j \end{cases}$$

Where M (i, j) is the probability of successfully reaching state S_j from S_i.

2.6.2 Parallel/Pipe-filter style

Under a sequential execution environment, only a single component is executed at a time. However, in a concurrent execution environment, components are commonly running simultaneously. The behavior of a software system with respect to the execution process can be modeled by using a Markov chain in which a state is defined by the execution of multiple components. In a parallel or a pipe-filter architectural style, multiple components can be executed concurrently, as shown in the dotted area of Figure 2.9(a). The difference between these two styles is that parallel computation is generally in multi-processors environment, whereas pipe-filter style occurs commonly in a single processor, multi-processes environment. Figure 2.9(b) is the state diagram of the parallel/pipe-filter architectural style, where the executions of the components C2 to Ck-1 are congregated into the state S p1 which is an element of the parallel state set S p.

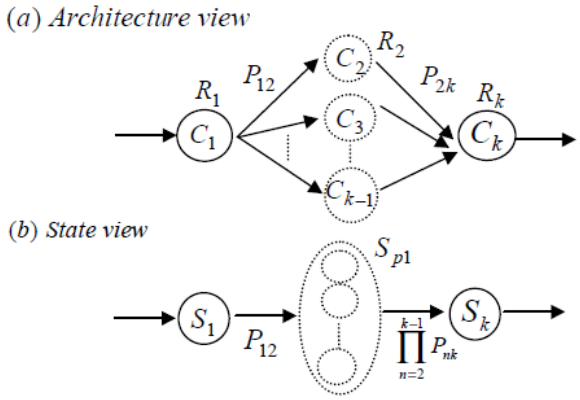

Figure 2.10: Parallel or pipe-filter style [26]

In Figure 2.9 (a), there are k components in which l=k-2 components are running concurrently into the same state; therefore, the total number of states is k-l+1. Because of the characteristics of parallel style, the transition probabilities from component C1 to components C2, C3…. and C_{k-1}, are all equal to P_{12}, which is now the transition probability from state S_1 to Sp_1. For convenience, we introduce $\{S_i\}$, which returns the row number or the column number of state variable S_i in a matrix. Entry M ($\{S_{p1}\}$, $\{S_k\}$) requires that all the components from C_2 to C_{k-1} in state S_{p1} perform successfully and finally reach state S_k. Because component reliabilities and

transition probabilities are all independent of each other, therefore the value of M ($\{S_{p1}\}$, $\{S_k\}$) is equal to $\prod_{n=2}^{k-1} R_n\ P_{nk}$, which is the product of all the component reliabilities in this state and the transition probabilities from components C_2, C_3,…, and C_{k-1} to component C_k , respectively. For k components, the transition matrix can be obtained:

$$\begin{cases} M(i,j) = R_i P_{ij},\ S_i \notin S_j \\ M(i,j) = \prod_{Cn\ in\ Si} R_n P_{nj},\ S_i \in S_p \quad \text{, for } 1<=I,\ j<=|S| \text{ And } 1<=n<=k \\ M(i,j) = 0,\ S_i \text{ cannot reach } S_j \end{cases}$$

CHAPTER 3
PROBLEM STATEMENT

Estimating software reliability during testing is useful in quantifying the quality of the software systems. However, such measurements applied late in the development process leave too little to be done to improve the quality of the software system in a cost-effective way. Reliability, an important attribute, is defined as the probability that the system performs its intended functionality under specified design limits. The first problem identified is that reliability models must be built to predict the system reliability at the initial phases of development process, and specifically when the implementation artifacts are unavailable.

A number of architecture-based software reliability models have been proposed by researchers which are used for predicting reliability. One of the categories of reliability models is state based model for terminating applications which uses control flow graph to represent the architecture of the system. Many architectural styles such as batch-sequential/pipeline, parallel/pipe-filter, call-and-return, and fault tolerance styles have already been implemented but they cover only the general aspect of the overall architecture.

There is a need to compare state based reliability models to find the limitations of model from the point of view of architectural style. Next, there is a need to enhance the style based architectural model to overcome the limitations of existing state based reliability model and incorporating design metrics into the software architecture evaluation.

CHAPTER 4
PROPOSED SOLUTION & VALIDATION

This section covers the process of finding a solution to the problem stated and achieves the objectives. Moreover, it validates the solution with a real time example. Firstly, the comparison parameters for state based models are defined and a comparison chart is prepared comparing all the models. Secondly, the solution is validated.

4.1 Comparison of State Based Models

First of all State Based Reliability Models are compared on the basis of the following comparison parameters:

- **Architecture:** Software architecture describes the structure of software at an abstract level consists of a set of components, connectors and configurations. This is the manner in which the different components of the software interact, and is given by the inter-modular transition probabilities. The architecture may also include information about the execution time of each component [19]. Interaction occurs only by transfer of execution control. In the case of sequential software, at each instant, control lies in one and only one of the components. In architecture–based approach, one must model the interaction of all components. In well designed system, interaction among components is limited. During the early phases of software life cycle, each component could be examined to find with which components it cannot interact [13].

- **Failure behavior:** The failure behavior of the components and of the interfaces between the components, is specified in terms of the probability of failure (or reliability), constant failure rate or time dependent failure intensity. The failure behavior is defined and associated with the software architecture. Failure can happen during an execution period of any component or during the control transfer between two components. The failure behavior of the components and of the interfaces between components can be specified in terms of their reliabilities or failure rates [19].

- **Solution method:** The solution method is a set of equations used to calculate number of failures or failure rate and system reliability. Further, it also forms a base for sensitivity analysis and optimization.

- **Sensitivity analysis:** Sensitivity analysis is the study of how the variation (uncertainty) in the output of a mathematical model mathematical can be apportioned, qualitatively or quantitatively, to different sources of variation in the input of the model. It is a technique for systematically changing parameters in a model to determine the effects of such changes. It is to improve the system reliability most effectively by evaluating the sensitivity of the system reliability with respect to that of a module.

- **Applications:** The applications of a reliability model simply define the type of domain or area covered by the model. In other words, it includes the applications on which a particular model is applied.

- **Optimization:** It means solving problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set. Optimization, or mathematical programming, refers to choosing the best element from some set of available alternatives.

Table 4.1: Comparison Chart

	Cheung Model [17]	Kubat Model [18]	Gokhale et al. [19]
Architecture	Uses absorbing DTMC. Transition probability matrix $P = [p_{ij}]$	Follows a DTMC. Architecture model for each task is SMP.	Uses absorbing DTMC. Coverage analysis tool ATAC is used for finding transition probabilities. Expected time in a module is calculated.
Failure Behavior	Module fails independently. Failure of a module depends on its probability to perform a task.	Failure intensity of a module *i* is λ_i.	Showed by NHPP model using time dependent failure intensity along with block coverage measurements and

			failure density approach.
Solution Method	Composite Method	Composite Method	Hierarchical Method
Sensitivity Analysis	√	×	×
Applications	Used to develop an effective testing strategy	Used in applications which use multiple modules with multiple tasks and repetitive usage	Testing critical applications
Optimization	×	√	×

DTMC = Discrete Time Markov Chain
SMP = Semi-Markov Process
ATAC = Automatic Test Analyzer in C

4.2 Client Server based Architectural Style Model

A number of architectural based models have been proposed. These models are classified according to the several different classification systems. There are basically two types of models based on the type of application; they are, for continuously running applications and for terminating applications. In this work, the models for only terminating applications are considered.

For a system with a complex architectural environment, this reliability model needs to be further refined to take the characteristics of different architectural styles into account. The architecture-based reliability model to compute the reliability of heterogeneous software systems may be comprised of various architectural styles. Some of them are already been implemented, we will focus here on Client Server based architectural style.

In order to utilize the Markov model, a transformation for each architectural style from an architecture view to a state view is introduced. Furthermore, based on the transformed state view, the transition matrix M (as shown in (1)) can be refined to obtain the style-based software reliability. The details of client server style is described as the following, where, Ri represents the reliability of component Ci , and Pij represents the probability of transition from component Ci to its successor component C j . To take the connector reliability into account, Pij could be adjusted as

the original transition probability multiplied by the reliability of the corresponding connector.

In the client server style, the client component may request some services which are provided by the server components. A component can act as a server to one component but may act as a client to other component. Therefore, the components may act as client or server at the same time. Also, the reliability of connectors between the client and server component is important. Keeping this in mind the state view in Figure 4.1(b) is obtained by one-to-one mapping to the architecture view shown in Figure 4.1(a)

(a) Architecture view (b) State view

Figure 4.1: Client Server Architectural Style

The entry M (1, 3) is equal to R1P13, which is the reliability of state S1 multiplied by the transition probability from S1 to S3. Likewise, the entry M (2, 1) can be computed as the reliability of state S2 multiplied by the transition probability from S2 to S1. The most important entry is M (1, 2) which only considers the transition probability from S1 to S2 without considering the reliability of state S1. This is because state S1 is the server. Therefore, the reliability of state S1 only needs to be considered when it acts as a client to another state.

Assuming there are k components, the total number of states is therefore k. The transition matrix M can be constructed as follows:

$$\begin{cases} M(i, j) = R_i P_{ij}, & \text{Si can reach Sj, when Si is client} \\ M(i, j) = P_{ij}, & \text{Si can reach Sj, when Si is server} \\ M(i, j) = 0, & \text{Si cannot reach Sj} \end{cases}$$

4.3 Example

The following example is used to demonstrate how to estimate the reliability of a system consisting of heterogeneous styles. Let Figure 5.2 be the directed graph representing the architecture view of a software system with seven components, where C_1 represents the input component and C_7 represents the output component.
The reliability of the components is shown in the following:

R1=0.999 R2=0.985 R3=0.990
R4=0.995 R5=0.980 R6=0.975
R7=0.980

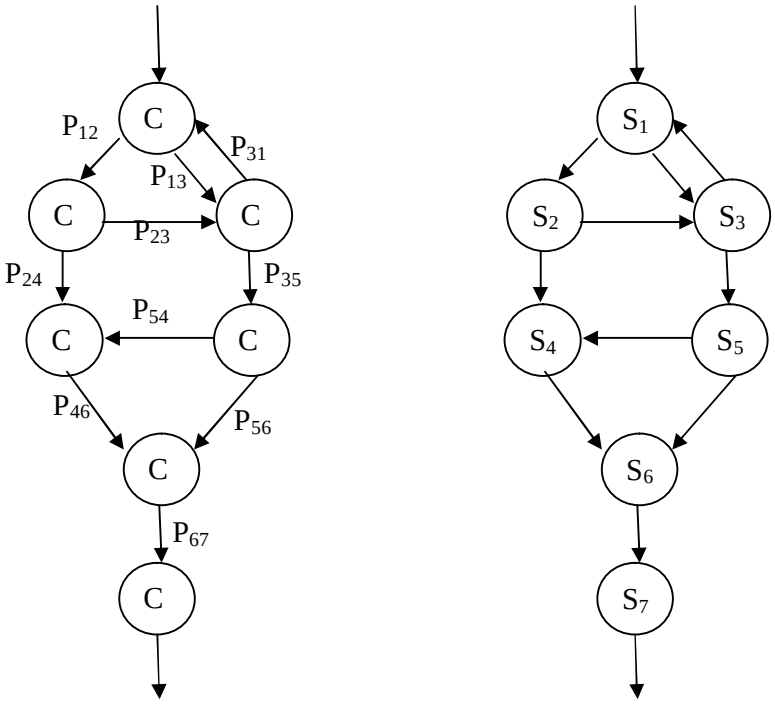

Figure 4.2: Architecture view & State view

To simplify the model, we assume the transition probabilities Pij between the components Ci and C j have already considered the connector reliabilities.

P1, 2=0.30 P1, 3=0.70 P2, 3=0.75
P2, 4=0.25 P3, 1=0.60 P3, 5=0.40
P5, 4=0.34 P5, 6=0.66
P4, 6= P6, 7=1.00

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_1	0	0.2997	0.7000	0	0	0	0
S_2	0	0	0.7388	0.2463	0	0	0
S_3	0.6903	0	0	0	0.3960	0	0
S_4	0	0	0	0	0	0.9950	0
S_5	0	0	0	0.3332	0	0.6468	0
S_6	0	0	0	0	0	0	0.9800
S_7	0	0	0	0	0	0	0

n=6

|I-M| = -0.0042

|E| = 0.004

$T(1, \{S_7\}) = (-1)^{n+1} \frac{|E|}{|I-M|} = 0.9527$

Overall System Reliability R

R = T (1, {S7}) × R7 = 0.9337

4.4 Validation of Proposed Reliability Model

We measured the reliability of the following OnlineAccMgr. The OnlineAccMgr is residing on internet-service host of a bank, providing online banking facilities. The model is motivated by the functionality provided by the online facility of one of the major Australian banks. In Figure 5.3 the provided protocol is shown. Also shown is an (estimated) usage profile, denoted by the probabilities given in squared brackets for each state transition. After the user logs in to the system (which includes the possibility of two failed trials), the system lists all accounts of the user with their balances. For a selected account, a pre-defined number of most recent transactions are listed. The user can scroll for further transactions and look at details of transactions, such as their receiver, ID, etc. At any time, the users can logout. When leaving the transaction-details view, the user can then scroll through the transactions of the selected account. The user also can select other accounts and proceed to inspecting their transactions without logging out [14].

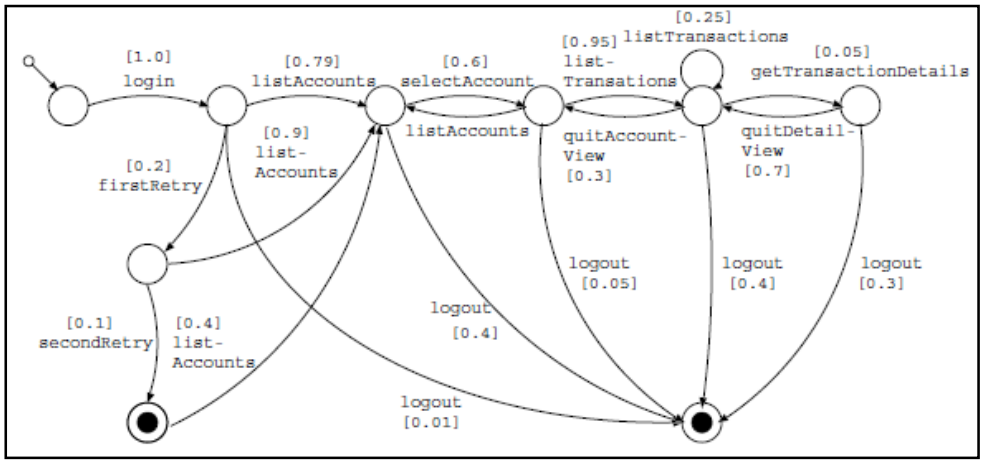

Figure 4.3: Transition diagram of the onlineAccMgr [14]

The transition probabilities P_{ij} between the components are shown in Figure 5.3. The reliabilities of the connectors are included in the transition probabilities. The reliability of the components are considered to be one, assuming that the component does not perform any failure.

Therefore, the transition matrix is given by

$$S = \begin{pmatrix} 0 & 0.9999 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.9495 & 0 & 0 & 0.0499 & 0 \\ 0 & 0 & 0 & 0.9995 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 09999 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.7999 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

n=6

|I-M| = 0.00797

|E| = 0.008

$T(1, \{S_7\}) = (-1)^{n+1} \frac{|E|}{|I-M|} = 0.9969$

Overall System Reliability R

R = T (1, {S7}) × R7 = 0.9969

The Predicted overall system reliability [14] is 0.9946.

Table 4.2: Predicted versus, measured reliability for OnlineAccMgr

Service	Predicted Reliability	Measured Reliability	Difference	Error (%)
onlineAccMgr	0.9946	0.9969	0.00013	0.013

Here lower error percentage indicates that the proposed reliability model is capable of measuring the reliability with high accuracy.

CHAPTER 6

CONCLUSION & FUTURE WORK

The reliability of a system depends on the reliability of the component, connectors and transition probability. This thesis extends the style based reliability model. This work adds client server architectural style to the heterogeneous reliability model based on the characteristics of various architectural styles. Further it provides example to support the proposed extension and then validates it with a real time example.

As now a days the systems are developed using modules or components the reliability needs to be measured while developing and integrating these components in developing a system so as a future work the model can be extended to other architectural styles across various application domains, and incorporating design metrics into the software architecture evaluation.

Further there is no common tool which can predict the reliability by just providing the reliabilities and transition probability and can identify the heterogeneous architectural styles automatically.

References

[1] P. C. Clements, “From Subroutines to Subsystems: Component-Based Software Development”, American Programmer, vol. 8, No. 11, November 1995.

[2] G. T. Heineman, W. T. Councill, “Component-Based Software Engineering: Putting the Pieces Together”, Addison-Wesley Professional, 1st ed., May 2001.

[3] C. V. Ramamoorthy, F. B. Bastani, “Software Reliability — Status and Perspectives”, IEEE, Trans. Soft. Eng., SE-8, No. 4, July 1982, pp. 354 – 371.

[4] S. Mahmood, R. La and Y. S. Kim, “A survey of component-based system quality assurance and assessment”, IET Software 1 (2), 2005, pp. 57–66.

[5] Y. Zhu, J. Gao, “A method to calculate the reliability of component-based software”, In Proceedings of 12th Asian Test Symposium, (ATS 2003), 2003, p. 488–491.

[6] S. Yacoub, B. Cukic, H. Ammar, “Scenario-based reliability analysis of component-based software”, Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE’99), 1999, pp. 22–31.

[7] M. R. Lyu, “Software Reliability Engineering: A Roadmap”, Future of Software Engineering, May 23-25, 2007.

[8] M. Shaw, D. Garlan, “An Introduction to Software Architecture”, Technical Report CMU/SEI-94-TR-21, ESC-TR-94-21, CMU Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1994.

[9] A. Wood, “Software Reliability Growth Models”, Tandem Computers, TR 96.1, September 1996, Part No. 130056.

[10] C. Smidts, R.W. Stoddard, M. Stutzke, “Software Reliability Models: An Approach to Early Reliability Prediction”, IEEE Transactions on Reliability, 47(3), 1998, pp. 268- 278.

[11] S. S. Gokhale, P. N. Marinos, K. S. Trivedi, "Important Milestones in Software Reliability Modeling", Proceedings of Software Engineering and Knowledge Engineering (SEKE '96), Lake Tahoe, NV, 1996, pp. 345-352.

[12] S. Gokhale, W. E. Wong, K. S. Trivedi, and J. R. Horgan, “An Analytical Approach to Architecture-Based Software Reliability Prediction”, IEEE Int. Computer Performance and Dependability Symposium, Sept. 1998, pp. 13-22.

[13] K. Goseva-Popstojanova, A.P. Mathur and K.S. Trivedi, “Comparison of Architecture-Based Software Reliability Models,” Proc. Int. Symp. Software Reliability Eng., 2001, pp. 22-31.

[14] R. Reussner, H. Schmidt, I. Poernomo, “Reliability prediction for component-based software architectures”, In Journal of Systems and Software, 66(3), Elsevier Science Inc, 2003.

[15] K. Goˇseva–Popstojanova and K. Trivedi, “Architecture–based approach to reliability assessment of software systems”, Performance Evaluation, Vol. 45, 2001, pp. 179-204.

[16] S. S. Gokhale, K. S. Trivedi, “Analytical models for architecture-based software reliability prediction: A unification framework”, IEEE Trans. on Reliability, vol. 55, issue 4, 2006, pp. 578-590.

[17] R. C. Cheung, “A user-oriented software reliability model”, IEEE Transactions on Software Engineering, SE-6, No. 2, 1980, pp. 118-125.

[18] P. Kubat, “Assessing Reliability of modular software”, Oper. Res. Lett. 8, 1989, pp. 35-41.

[19] S. Gokhale, W.E. Wong, K. Trivedi, J.R. Horgan, “An analytical approach to architecture based software reliability prediction”, Proceedings of the Third International Computer Performance and Dependability Symposium (IPDS’98), 1998, pp. 13–22.

[20] W. Wang, Y. Wu, M. Chen, “An architecture-based software reliability model”, Proceedings of the Pacific Rim International Symposium on Dependable Computing, 1998, pp. 143–150.

[21] L. Bass, P. Clements, and R. Kazman, “Software Architecture in Practice”, Addison Wesley Longman, Inc., 1998.

[22] M. Shooman, “Structural models for software reliability prediction”, Proceedings of the Second International Conference on Software Engineering, 1976, pp. 268–280.

[23] S. Krishnamurthy, A.P. Mathur, “The estimation of reliability of a software system using reliabilities of its components”, Proceedings of the Eighth International Symposium on Software Reliability Engineering (ISSRE’97), 1997, pp. 146–155.

[24] S. Yacoub, B. Cukic, H. Ammar, “Scenario-based reliability analysis of component-based software”, Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE’99), 1999, pp. 22–31.

[25] W. Wang, Y. Wu, M. Chen, “An architecture-based software reliability model”, Proceedings of the Pacific Rim International Symposium on Dependable Computing, 1999, pp. 143–150.

[26] M. Shaw, “Procedure calls are the assembly language of system interconnection: Connectors deserve first class status”, Proceedings of the Workshop on Studies of Software Design, May 1993.

[27] H. Pham, “Software Reliability”, Springer-Verlag, 2000.

[28] D. Garlan, M. Shaw, “An Introduction to Software Architecture”, Carnegie Mellon University, Pittsburgh, PA, 1994.

[29] A. L. Goel, “Software Reliability Models: Assumptions, Limitations, and Applicability”, IEEE, Trans. Soft. Eng., SE-11, No. 12, Dec. 1985, pp. 1411 – 1423.

[30] C. A. Asad, M. I. Ullah, M. J. Rehman, "An approach for software reliability model selection," COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSACV4), IEEE Computer Society, 2004, pp. 534-539.

[31] N. G. Leveson , C. S. Turner, “An Investigation of the Therac-25 Accidents”, Computer, Vol. 26, No. 7, July 1993, pp.18-41.

[32] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture”, ACM SIGSOFT Software Eng. Notes, Vol. 17, No. 4, October 1992, pp. 40-52.

[33] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, “A component and message based architectural style for GUI software”, IEEE Transactions on Software Engineering, June 1996.

[34] R. Meunier, “The Pipes and Filters Architecture”, Proceedings of the First Conference on Pattern Languages and Programming, Addison-Wesley, 1994.

[35] G. Booch, “Object-Oriented Analysis and Design with Applications”, Addison-Wesley, Reading, MA, 1994.

[36] A. L. Goel, “Software Reliability Models: Assumptions, Limitations, and Applicability”, IEEE, Trans. Soft. Eng., SE-11, No. 12, Dec. 1985, pp. 1411 – 1423.

List of Paper Published/Communicated

- Arpita Sharma, Ms. Shivani Goel, "Comparison of Architecture based Reliability Model", International Journal of Advanced Engineering Sciences and Technologies , Vol. 7, No. 2, 2011 (Accepted)
- Arpita Sharma, Ms. Shivani Goel, "Client Server Architectural Style in Reliability Model", Research Journal of Computer Systems and Engineering (Communicated)