HIGH AVAILABILITY

HIGH AVAILABILITY AND SOFTWARE ARCHITECTURE

By
Rongshu Yi, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (2007) McMaster University
COMPUTING AND SOFTWARE Hamilton, Ontario

TITLE: HIGH AVAILABILITY AND SOFTWARE ARCHITECTURE

AUTHOR: Rongshu Yi, B.Eng. (McMaster University)

SUPERVISOR: Dr. Tom Maibaum

NUMBER OF PAGERS: xii, 98

Abstract

As one of the main objectives of Enterprise Business Applications, high availability is becoming more and more important and is a widespread concern of customers, architects and developers. But, until now there are still no models for computing and evaluating accurately the value of a system's availability, and no any feasible and systematic methods to improve the availability of a software system, which may be made up of several patterns, and may be very complex. PBSA(Part Based Software Architecture), as a new analytical model for predicting a complex software system's availability, is based on the availability of each component, the properties of some main architecture patterns, and the architectural structure of the software system itself. It uses a Markov model to translate the evaluation process of a system's availability into a Markov Transfer process, and uses the properties of Markov Chains to calculate and predict the probability of the whole system's availability. Then based on the model, a sensitivity rank for architecture components and factors can be given, and systematic strategies for improving the availability property of a complex application system can be given. As one of the strategies for improving system availability, a new component, named the HA(high availability) manager component, is defined and introduced, together with some idea of an HA Hierarchy.

Acknowledgements

First of all, I would like to express my sincere thanks and deep appreciation to Dr. Tom Maibaum, my supervisor, for his constant support and encouragement. He shared so many great ideas with me and carefully corrected my mistakes and typos. This thesis would not be what it is without him. He is one of the nicest professors I have ever met.

I also want to express my appreciation to Dr. Doug Down, who gave me lots of great suggestions for the PBSA design. I am grateful to Dr. Emil Sekerinski for reviewing this thesis and giving me valuable feedback and suggestions.

Of course, I am very thankful to my wife for her support and endless love.

Hamilton, Ontario, Canada — Rongshu Yi
April, 2007

Contents

List of Figures

Introduction

Early quality prediction at the architecture design stage is highly desired by software managers and developers, as it provides a means for making design decisions and thereby facilitates effective development processes [17]. If a design flaw, especially in a large scale software system, is detected in the implementation or testing stage, it is more difficult and more expensive to make changes and corrections than at the architecture design stage. We claim that a system with low availability without satisfying the quality requirement of a system is such a big flaw. Software architecture analysis aims at investigating how an architecture meets its quality requirements [29], 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 [3, 13, 16], consists of a set of *components*, *connectors*, and *configurations*. Furthermore, a repeatable pattern that characterizes the configurations of components and connectors of a software architecture is considered an architecture pattern [4, 18]. Many architecture patterns have been identified, with new ones continuously emerging [4, 18, 19]. Thus, an architect is faced with the challenges of selecting suitable patterns, or modeling the configuration of selected patterns, for designing the architecture for a given software specification. A method or model to predict or evaluate the availability of a complex software system, which may be composed of heterogeneous patterns, can certainly provide a means through which designers can reconfigure the architecture to best fit their availability quality requirements.

Although high availability is becoming more and more important, and is a widespread concern of customers, architects and developers, until now there are still no concrete and quantative models for computing accurate value of a system's availability, and no feasible and systematic methods to improve the availability of a large scale software system, which may be made up of several different patterns, and may be very complex. Some qualitative models have been introduced for architecture patterns during the past years. But, as we know, many large scale systems are composed of several heterogeneous patterns or styles, and may be very complex, so using the qualitative model is very hard for practitioners and architects to decide how to improve the availability of such a very complex software system. In this thesis, we present an analytical model PBSA(Part Based Software Architecture), for quantitatively predicting and evaluat-

ing the availability of a software architecture-based system , based on the availability of each component, the properties of some main architecture patterns, and the overall architectural structure of the system. Then based on the model, we define a method to build a sensitivity rank for all components and availability factors, and then give some feasible and systematic strategies for improving the availability property of any general application system, which may be made up of different patterns and may have different contexts, by spending the least amount of money to get the most efficient availability contribution. The model is based on Markov chain properties and the transformation from an Architecture view to a state view, and the basic concepts of availability. In addition, a new component, named HA (high availability) manager component, is introduced, as a tool for helping to improve the availability of the whole system, and some idea for an HA Hierarchy are given as well.

In Chapter 1, we give some background for availability and software architecture and architecture patterns (styles), and give the relationship between availability and reliability. In Chapter 2, we present some qualitative analytical models for availability quality analysis. In Chapter 3, we introduce a new availability analytical model based on parts, and on software architecture, which may be composed of different patterns or styles. In this Chapter we also introduce an example to show how to compute the availability of an existing system architecture. We introduce a procedure for reducing a complex large scale system to a pure part normal architecture system. In Chapter 4, we present the strategies for choosing the most significant components and factors to improve the availability of individual components and how to use the strategies together to improve the availability of a whole system. Using the same example, we show that the availability is improved significantly by adopting the strategy.

1 Background

1.1 Availability

All kinds of software systems, especially real time and embedded systems, are now a central part of our lives. Available functioning of these systems is of paramount concern to the millions of users that depend on these systems every day. Unfortunately, most such systems still fall short of expectation of availability. There are still lots of this or that kind of failure. Before discussing the notion of availability, we would like to first introduce the characteristics of failures.

1.1.1 Failure Characteristics

We can first classify failures into two groups, one of which is hardware failure; the other is software failure.

Hardware Failure

Hardware failures can be typically characterized by a bath tub curve[1]. The chance of a hardware failure is high during the initial life and during the end life of the module. But the failure rate during the middle of the life of the product is fairly low. So the hardware vendors offer a guideline of failure probability for different terms during the life cycle of their products.

Hardware failures during a product's life can be attributed to the following causes:

- *Design failures* This class of failures take place due to inherent design flaws in the process of system design. In a well designed system, this class of failures should make a very small contribution to the total number of failures.

- *Random failures* Random failures can occur during the entire life of a hardware module. These failures can lead to system failure. This class of failures can not be avoided totally, but we can find someway to make the lost be minimal. HA components and Redundancy are such ways, provided to recover from this class of failures.

- *Infant mortality* This class of failures cause newly manufactured hardware to fail. This type of failures can be attributed to manufacturing problems like poor soldering, leaking capacitor etc. These failures should not be present in systems, after leaving factory, as these faults will show up in factory system burn in tests.

[1]The curve represents the number of failure times over a time unit.

- *Wear out* Once a hardware module has reached the end of its useful life, degradation of component characteristics will cause hardware modules to fail. This type of faults can be weeded out by preventive maintenance of hardware.

- *Unsuitable use* This class of failures cause hardware to fail by not giving it a suitable environment. For example, the power system of a lab is unsuitable or the air conditioner of the lab is not sufficient. This class of failures can be avoided by giving hardware suitable environment, and meanwhile redundancy can be used to decrease the lost.

Software Failures

Software failures can be characterized by keeping track of software defect density[2] in the system. This number can be obtained by keeping track of historical software defect history. Defect density will depend on the following factors:

- Complexity of the software.
- Size of the software.
- Experience of the team developing the software.
- Percentage of code reused from a previous stable project.
- Rigor and depth of testing before product is shipped.
- Software process used to develop, design and code.
- Management of a software development life cycle.

Defect density is typically measured in number of defects per thousand lines of code (defects/KLOC).

Here are some examples of why Software failures occur:

- Heap space runs out $\longrightarrow$ memory can not be allocated.
- Server (or client) goes down $\longrightarrow$ message cannot be sent.
- File or disk space runs out $\longrightarrow$ file cannot be written.
- Input is in wrong format$\longrightarrow$ data cannot be read or transferred.

[2]Number of defects per unit of lines in source code

- Interpreter passed a program with a run-time error $\longrightarrow$ cannot interpret.
- Number is 0 $\longrightarrow$ cannot divide.

We have reviewed the concept of failures and know that there are two groups of failures, hardware failure and software failure. And we know what they are. Then, we now present the idea of availability.

1.1.2 What is Availability

Availability refers to a level of service provided by applications, services, or systems. Highly available systems have minimal downtime, whether planned or unplanned. Availability is frequently expressed as the percentage of time that a service or system is available, for example, 99.9% for a service that is unavailable for 8.75 hours per year.

To define it more clearly and formally, we will introduce some notations first.

1. *MTBF* Mean Time Between Failures *MTBF*, as the name suggests, is the average time between failures of hardware or software modules. It is the average time a manufacturer estimates before a failure occurs in a hardware module or the average time between failures of a software system module or component or the entire software system [21].

   MTBF for hardware modules can be obtained from the vendor for off-the-shelf hardware modules. *MTBF* for inhouse developed hardware modules is calculated by the hardware team developing the board.

   MTBF for software can be determined by simply multiplying the defect rate with *KLOCs*, executed per second. (We will not talk more about this notion, because this is not the most sensitive factor affecting the entire system availability).

2. *MTTR* Mean Time To Repair (*MTTR*), is the time taken to repair a failed hardware module or to recover from a failed software module, component or software system. In an operational system, repair generally means replacing the hardware module. Thus hardware *MTTR* could be viewed as mean time to replace a failed hardware module. It should be a goal of system designers to allow for a high *MTTR* value and still achieve the system availability goals. In a redundant environment, the *MTTR* for hardware is still the *MTTR* for a single component, and we do not compute the *MTTR* of a redundancy

component group, but actually there is *MTTR* in a redundancy component group, we ignore it here, just because it may be small enough for us to ignore it, if we use correct HA strategies.

The *MTTR* for a software module can be computed as the time taken to reboot or maintain after a software fault is detected. The main goal of system designers, especially developers, should be to keep the software *MTTR* as low as possible.

Now we can define the availability of a module[3] as follows.

Definition 1.1 Availability *of a module is the percentage of time when the module is operational.*

So, the availability of a hardware/software module can be obtained by the formula given below:

$$A = \frac{MTBF}{MTTR + MTBF} \tag{1}$$

Availability is typically specified in nines notation. For example 3-nines availability corresponds to 99.9% availability. A 5-nines availability corresponds to 99.999% availability.

According to the definition of availability, we know there is a corresponding notion of *Downtime*. Downtime per year is a more intuitive way of understanding the availability. The table below compares the availability and the corresponding downtime.

Availability	Downtime
90% (1-nine)	36.5 days/year
99% (2-nines)	3.65 days/year
99.9% (3-nines)	8.76 hours/year
99.99% (4-nines)	52 minutes/year
99.999% (5-nines)	5 minutes/year
99.9999% (6-nines)	31 seconds/year

1.1.3 Why we need high availability

In the above section, we defined what the availability is. Now, somebody may have some idea of improving availability by using redundancy or by using other means to decrease *MTTR* or increasing *MTBF*. All these methods clearly first occurred in our mind when we think of availability. Except for how to use these methods, we may think it is expensive of the

[3]The module here can be a process, a component or a system.

methods themselves, and we may doubt whether it is worthy doing them. Let us think about a scenario, a simple server crash, and what it costs our company. The following is a list of what could happen, in sequence:

1. A company uses a server to access an application that accepts orders and does transactions. (This translates to 'how the company collects cash from customers'.)

2. The application, when it runs, serves not only the sales staff, but also three other companies who do business-to-business (B2B) transactions. The estimate is that, within one hour's time, the peak money made exceeded 30 million dollars.

3. The server crashes or the application goes down and you do not have a High Availability solution in place.

4. This means no failover, redundancy, or load balancing exists at all. It simply fails.

5. It takes you (the Systems Engineer) 5 minutes to be paged, but about 15 minutes to get on site. You then take 40 minutes to troubleshoot and resolve the problem. One hour's time is very conservative.

6. The company's server is brought back online and connections are reestablished. The system is tested and deemed physically and logically fit.

Everything appears functional again. The problem was simple this time-a simple application glitch that caused a service to stop and, once restarted, everything was okay.

Now, the problem with this whole scenario is this: although it was a true disaster, it was also a simple one. The systems engineer happened to be nearby and was able to diagnose the problem quite quickly. Even better, the problem was a simple fix. This easy problem still took the companies' shared application down for at least one hour and, if this had been a peak-time period, over 30 million dollars could have been lost, and even more, if there is a lots for other business vendors, then according to the contract, your company may pay the compensation for the lots. So now, we may wish we had that high availability solution. How much money would it take for the company to lose before we paid for the redundancy, the staff and their lunches for a year by being proactive? High Availability is based on proactive thinking. We are *planning* for disaster so we will not have to *react* and regret it once it occurs.

Another issue that could result in harm from the no protect from disaster is losing customer or vendor faith in your company. The companies you connect to and do business with as well as your own clientele may start to lose faith in your ability to serve them if your web site is not accessible or defaced, your database is corrupted, your ERP application is not accessible and holding them up from doing business. This could also cost you revenue and the possibility of acquiring new clients moving forward. People *talk* and the uneducated could take this small glitch as a major problem with your company's people, instead of the technology.

Let us look at this scenario again, except with a High Availability solution in place:

1. A company uses a Server to access an application that accepts orders and does transactions.

2. The application, when it runs, serves not only the sales staff, but also three other companies who do business-to-business (B2B) transactions. The estimate is, within one hour's time, the peak money made exceeded 30 million dollars.

3. The server crashes, or the application fails, but you do have a Highly Available solution in place. (Note, at this point, it does not matter what the solution is and what kind of redundancy you added into the service.)

4. Server and application are redundant, so when a glitch takes place, the redundancy spares the application from failing.

5. Customers are unaffected. It is transparent for them that our system has a switching operation, according to our high availability strategy. Business resumes as normal. Nothing is lost and no downtime is accumulated.

6. The one hour cost we saved for our business in downtime, is enough for us to pay for the entire Highly Available solution we implemented.

With a plan or policy in place, planning for proactive design and use of redundant and resilient services can help us to avert most disasters.

The days when a daily backup provided adequate protection for a business's critical processes are long gone. Today, downtime, any downtime, translates to real and significant costs, in lost productivity because employees can not work, lost business because orders can not be taken, and customer dissatisfaction as our downtime translates to their downtime. A recent study of 80 large organizations by *Infonetics Research* found that

overall downtime costs averaged an astounding 3.6% of annual revenue. Can you afford that kind of bite out of your margins? Well over half of all downtime is unplanned. Of that, 80% is due to software and user errors: unknown and irreproducible bugs, accidental deletion or configuration changes, batch jobs corrupting important data, and errors from the use of hard-to-understand management tools [37]. More importantly, many of these occur at predictable times, for example, during database maintenance or batch processing runs. Hence, we need high availability for our critical systems. But how to get it? That is a problem of planning first.

1.1.4 Plan a high availability system

To plan a high Availability System, we should first get clear the requirement of availability.

Defining the Availability Requirement

The availability of a service is a complex issue that spans many disciplines. Many different approaches can be taken to deliver the required levels of availability, each with their own cost implications.

However, availability requirements can frequently be expressed in relatively simplistic terms by the customer and without a full understanding of the implications. This situation can lead to misunderstandings between the customer and the IT organization, inappropriate levels of investment, and ultimately to customer dissatisfaction through inappropriate expectations being set.

One expressed requirement for 99.9 percent availability can be different from another requirement for 99.9 percent. One requirement may discuss the availability of the hardware platform alone, and the other requirement may discuss the availability of the complete end-to-end service. Even the definition of complete end-to-end service availability can vary greatly. It is important to understand exactly how any availability requirements are to be measured. For example:

- If all hardware and software on the primary server is functioning correctly and user connections are ready to be accepted by the application, is the solution considered 100 percent available?
- If there are 100 users but 15 percent of them can not connect because of a local network failure, is the solution still considered 100 percent available?
- If only one user out of the 100 users can connect and process work, is only 1 percent available?

- If all 100 users can connect but the service is degraded with only two out of three customer transactions being available, or performance is poor, how does this affect availability measurements?

The period over which availability is to be measured can also have a significant effect on the definition of availability. A requirement for 99.9 percent availability over a one-year period allows 8.8 hours of downtime. A requirement for 99.9 percent availability over a rolling four-week window only allows 40 minutes downtime in each period.

It is also necessary to identify and negotiate periods of downtime for planned maintenance activity, service pack, and software updates. The amount of planned downtime that can be tolerated has a significant effect on the definition of availability requirements.

In [30], a scenario-based method for availability requirement definition is shown. There is a very popular approach to define the availability requirement for a software system today. This will be discussed in detail in Chapter 2.

Planning for High Availability

To ensure that systems are available for requests when called upon, you have to plan for the failure of those same resources that are being requested. Hard drives have $MTBF$ (mean time between failure), viruses go undetected. There are many ways that a disaster can happen. The only way to keep systems 'available' to those who request them is to deploy those systems in a manner that will keep them available to those requests under these scenarios.

Either with disk arrays or clustered servers, planned redundancy is always a good bet to take when planning for high availability. In this thesis we will look at what you need to know to plan for a high availability solution that will keep services online and available to those who need them, and depend on them. High availability takes some work and effort in the beginning. Taking the time to plan and design is the key to maximizing the possibility of a successful deployment. The same care must go into the design process. High availability design is often complex and requires knowing a great many areas within IT to get it right - or the team used to plan the high availability solution ends to be diverse. Call that, high availability assures uptime, uptime may be the core of the business, so we should consider the costs of implementing a highly available solution; one failure that causes your serious downtime may be all it takes to have paid for the solution in the first place.

So, to get high availability of a system, we should have a plan first.

To build a plan, we should first know exactly the requirement of availability. Then as we said in the introduction, if a design flaw, especially in a large scale software system, is detected in the implementation or testing phase, it is more difficult and more expensive to make changes and correction than at the architecture design stage. The low availability, which cannot satisfy the quality requirement of customers, of a system is such a great flaw. So, it is the responsibility of an architect to make the design decisions to satisfy the availability requirement at the architecture design stage.

We know a lot of approaches to improve high availability, but we do not know exactly how to. For example, we know redundancy can be used to improve the availability significantly, but which component or components should be chosen to use redundancy on? It is unnecessary and impossible to use redundancy on all components, especially on clients desktops, except if we have too much to spend. In addition, we should know before we make decisions at the design stage, which means are the best or most useful ways to improve the availability of a system or component. There are many ways we can take to improve the availability of a system, such as Redundancy, better maintainability, heartbeat monitor, Watchdog and so on. The best way is to choose the most sensitive components, use the most sensitive means and satisfy the requirement of availability using the least cost. In Chapter 3, we show how to calculate the entire system's availability using some models, based on component, architectural style(pattern), and architecture structure. Then, in Chapter 4, we indicate how to choose the most sensitive components and most sensitive means and how to improve a single component's availability and how to improve the entire system's availability.

So, we may have the following steps and **strategies** to build a plan for high availability:

1. Get the requirement for availability, based on the scenarios method (see Chapter 2).

2. Use the model of PBSA (Part Based Software Availability) to compute the availability of current system, based on the PBSA model and current architecture structure.

3. If the result of the availability computed above does not fit for the requirement, use availability improvement methods, i.e. choose the most sensitive component, and use the most sensitive means, or tune the architecture structure to improve the availability of the system. Replace the current architecture structure by the improved one, delete current component from the sensitivity order list (SOL).

4. Repeat steps 2 and 3, until the availability computed satisfies the requirement.

Before we present the PBSA model, we would like to give some foundational background for Architecture and Architectural style, and to avoid confusion, we would first compare availability with reliability.

1.1.5 Availability and Reliability

Availability is defined as the probability that the system is operating properly when it is requested for use. In other words, availability is the probability that a system is not failed or undergoing a repair action when it needs to be used. If not considered carefully, it might seem that if a system has a high availability then it should also have a high reliability. However, this is not necessarily the case.

Reliability represents the probability of components, parts and systems to perform their required functions for a desired period of time, without failure in specified environments and with a desired confidence [8]. Reliability, in itself, does not account for any repair actions that may take place, while it accounts for the time that it will take the component, part or system to fail while it is operating, or the probability of not failing while running. It does not reflect how long it will take to get the unit under repair back into working condition.

As we defined earlier in Chapter 1.1.2, availability represents the probability that the system is able to offer its function when it is called and the system is not failed or undergoing recovery. So clearly, it is not only a function of reliability, but also a function of maintainability.

Below is a relationship table between Reliability, Maintainability, and Availability by the eMagazine for the Reliability Professional (Issue 26, April 2003), in which an increase in the maintainability means a decrease in the fix time when a system fails.

Reliability	Maintainability	Availability
Constant	Decrease	Decrease
Constant	Increase	Increase
Increase	Constant	Increase
Decrease	Constant	Decrease

As we can see from the table, if the reliability is held constant, then even at a high value, this does not directly imply a high availability. As Maintainability decreases, or the time to repair increases, the availability decreases. Even a system with a low reliability could have a high availability if the time to repair($MTTR$) is very short.

So, we can conclude that to increase Availability, there are two ways we can go. The first one is to increase reliability, and the second one is to increase maintainability. We know that increasing reliability can increase $MTBF$, but the availability of a system is equal to

$$\frac{MTBF}{MTTR + MTBF}$$

Then, to increase availability, we could increase reliability and decrease $MTTR$ or at least keep it constant. That increasing Maintainability can decrease $MTTR$ is a well known idea [12,20].

1.2 Software Architecture and Quality Attributes

What is software architecture and how to make the quality attribute be one of the concerns an architect may consider when doing architecture design? These are concerns in building high availability systems.

1.2.1 Software architecture

What is software architecture? There are many many definitions of it, and we will choose one of the most popular and reasonable ones and also the most useful one for our use in PBSA. This definition is from [3], which represents a reasonably common view:

Definition 1.2 The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The key elements of this definition include:

- Architecture is an abstraction of a system or systems. It represents systems in terms of abstract components which have externally visible properties and relationships (sometimes called connections, although the notion of relationships is more general than connections, and can include temporal relationships, dependencies, uses relationships, and so on).

- Because architecture is about abstraction, it suppresses purely local information; private component details are not architectural. Just like an object in Object-Oriented Design (OOD), with its encapsulation, you can only see its interfaces.

- Systems are composed of many structures (commonly called views). Hence there is no such thing as **the** architecture of a system, and no single view can appropriately represent anything but a trivial architecture. Furthermore, the set of views is not fixed or prescribed. An architecture should be described by a set of views that support its analysis and communication needs. This does not mean we should use different views for one purpose, actually in this thesis, we would like to use the **structure view only** for architecture representation.

- Externally visible properties are those assumptions other elements can make about an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

Architecture plays a very important role in software system design and development. Although an architecture is a technical description of an engineering blueprint of a system, it affects everyone involved with the system. Each stakeholder of a system, customer, user, project manager, coder, tester, etc. is concerned with different characteristics of the system that are affected by its architecture. For example, a user is concerned that the system is usable, reliable and available; a customer is concerned that the architecture can be implemented on schedule and to budget; a manager is worried (in addition to cost and schedule) that the architecture will allow development teams to work largely independently, interacting in disciplined and controlled ways; a developer is worried about achieving all of those goals through coding; a tester wants to prove (or disprove) that these goals will be met. Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to communicate and comprehend large systems sufficiently to make the early decisions that influence both quality and usefulness.

Component and Connector

In an architecture, the main elements are runtime components (which are the principal units of computation) and connectors (which are the communication vehicles among components). Component-and-connector structures help answer questions such as: What are the major executing components and how do they interact? What are the major shared data stores? Which parts of the system are replicated? How does data progress through the system? What parts of the system can run in parallel? How can the system's structure change as it executes?

Components are abstract and have externally visible properties and relationships. One of the externally visible properties is the availability, which is the key property we are going to discuss and focus on in this thesis. The availability of a component is the probability that the component is available. It is the main foundation of the PBSA model.

So, what is the significance of the connectors? Do the connectors mean that the elements communicate with each other, send data to each other, control each other, synchronize with each other, use each other, invoke each other, share some information-hiding secret with each other, or some combination of these or other relations? What are the mechanisms for the communication? What information flows across the mechanisms, whatever they may be? All these are key questions the architect will pay attention to when he or she is designing an architect for the connection

between components. But we will neglect all these questions, and pay attention to the probability of successful transition from one component to another one. The mechanization of how they communicate with each other is not what we concern about, while the probability of the transition will be, because it may affect the entire system's availability.

In PBSA, we will view **a connector as a special component and even a special Part and say it has an availability, just as a component does.**

Architectural Structure

As described in [3], in a house, there are plans for the structure of the house, the layout of the rooms, for electrical wiring, plumbing, ventilation, and so forth. Each of these plans constitutes a "view" of the house. These views are used by different people. The electrician primarily uses the wiring view. The carpenter primarily uses the structural view. Each specialist uses their own view to achieve different qualities in the house. The carpenter is primarily interested in making the walls straight and square and in assuring that joists are of sufficient strength to support the floors. The electrician is primarily concerned with providing appropriate electrical capacity in convenient locations, and in doing so in such a way as to safeguard the house's occupants.

According to [3] and [16], common architecture views include:

- logical view
- code view
- development view
- concurrency (or process/thread) view
- physical (or deployment) view

The definitions and classifications are still being debated, but we should not care about that, because we will only use the logical or functional view for PBSA.

The logical or functional view is an abstraction of system functions and their relationships. The components of the functional view are: functions, key system abstractions, and domain elements. The connectors or relationships between components found in the view are dependencies and data flow. In this thesis, all the relationships will be presented as a transition from one component to the other, as in data flow. The dependencies, such as a call/reply will be described as a special relationship between two

components, and will be discussed in detail in the Chapter of architectural patterns. As we mentioned above, all connectors will be viewed as special components, so we can forget about connectors in this thesis.

An example of a purely logical or functional view is shown in Figure 1. This is an application of the Business Intelligence for a bank data warehouse.

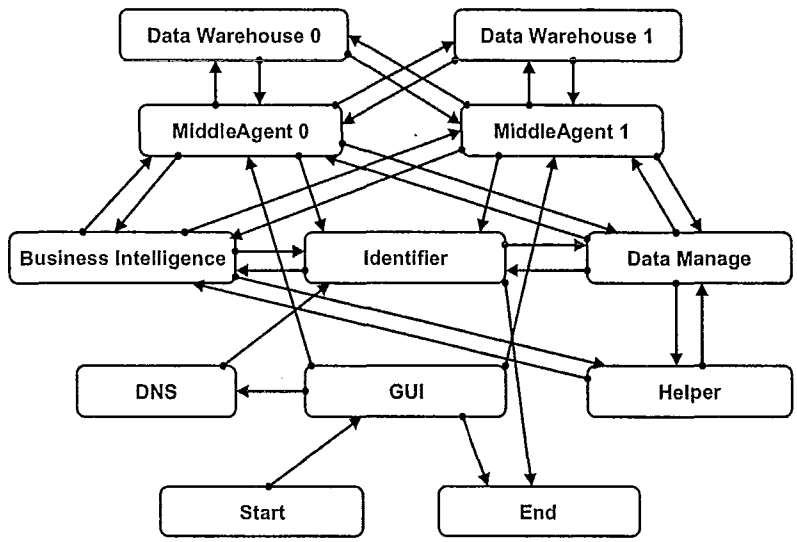

Figure 1: The Logical view of BI Bank Architecture

In this figure, we set all the components to be functional and let the connectors be some special components. The directed connection line indicates the data flow from one component to the other and indicates the control transition from one component to the other most of the time, except for the only connections between *Data Manage* and *Helper*, *Decision Analysis* and *Helper*, *Business Intelligent* and *Middle Agent0*, *Business Intelligent* and *Middle Agent1*, *Middle Agent0* and *Data Warehouse0*, *Middle Agent0* and *Data Warehouse1*,*Middle Agent1* and *Data Warehouse1*, *Middle Agent1* and *Data Warehouse0*. These connections are relationships, named *Call and Return*, *Parallel Computing*, *Redundancy*, which are some of the special architecture patterns, which we will present in next Chapter.

1.2.2 Software Architecture and Quality Attributes

In essence, software architecture design, like any other design process, focuses on making decisions, guided by requirements and the current environment. These decisions may involve:

- Separating one set of responsibilities from another component separation;
- Duplicating responsibilities if needed;
- Allocating responsibilities to processors (decomposition);
- Determining how discrete elements of responsibilities cooperate and coordinate (define the relationship between or among components)

At the architecture design level, we need not define much about the details of implementation and refinement of the components, but we should define clearly the quality attributes, for example, availability, security, performance and so on.

Let us consider various attributes at this level. Availability intuitively implies redundancy and other HA strategies for the most sensitive components. Performance depends on processes, their allocation to processors, communication paths between them, and other factors. Maintainability requires dependency chains. In each case, architecture design decisions are necessary to achieve the appropriate level for the respective attribute. In addition, these decisions require very little knowledge of functionality, especially for the quality of availability.

Now let us see how a software architecture decision can affect the quality attribute. Suppose we separate one collection of responsibilities from another. As a result of this decision, we can ask a number of quality attribute questions: What is the impact on modifiability? What are the availability implications? Does this separation have any effect on performance, security, or usability? All of these questions should be asked and, in some cases, answered as a result and effect of the decision. Thus, every decision (including the separation and the other main classes of decisions) embodied in a software architecture can potentially affect quality attributes. At least, each decision raises questions about its effect on the quality attributes and often these questions cannot be answered by examining the decision, but are answered in the context of additional decisions. We originally separated the collections of responsibilities for a reason; for example, we might have wanted to support later modifications or to allocate separate portions to different processors to increase performance. At some time, we might have wanted to separate computations to improve reliability. Our decision could have been for other reasons.

In each case, we can discuss how the decision supports those goals. And sometimes, one decision for benefiting a quality attribute, may hurt the other attributes, such as availability and performance. To improve availability, we may make a decision to use voting redundancy, presented

in Chapter 2, but this component may decrease the performance of the system.

To summarize, software architecture is closely coupled to how well a system achieves various quality attributes. The following are the main properties of the relationship between software architecture and quality attributes:

- Quality attributes constitute some of the main goals of software architecture design;
- One architecture decision may affect one or more quality attributes;
- A software architecture designed without considering the quality attributes will be fragile and easily lead to project failure;
- When we consider quality attributes, we cannot focus our attention only on one quality; most architecture design decisions are a tradeoff between different quality attributes.

But that does not mean that we will discuss all the quality attributes in this thesis. We will focus our attention on the availability attribute, without considering much about the other quality attributes, because the main purpose of this thesis is to give availability calculation models and availability improvement strategies. Other quality attributes will also be very important, depending on the requirement. If we are doing architecture design, and availability is the main object of the design, we can use this thesis to help; otherwise, we can use this thesis as one of the references, when evaluating the side effect of one architecture design decision, because we rank the availability sensitivity of components and factors, which affect the availability attribute.

In the following Chapter, we give some background about the relationship between architecture pattern and architecture and the relationship between architecture pattern and quality.

1.3 Architecture Pattern

1.3.1 What is Architecture Pattern

There are many kinds of definition of architectural patterns or architecture styles, just like the definition of software architecture. We choose the definition of Frank Buschmann [4],

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of a set of constraints on an architecture- on the element types and their pattern of interaction and these constraints define a set or family of architectures that satisfy them.

An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, including rules and guidelines for organizing the relationships between them.

We can think of architectural patterns as templates for concrete software architectures. They specify the system-wide or infrastructure level structural properties of an application, and have an impact on the architecture of their sub-systems. The selection of a software architectural pattern is a basic and fundamental design decision when developing a software application. The reason is obviously because the software architectural patterns are repeated and tested by many application designing architects, and there is no need to totally design by yourself for every concrete application.

But a software architecture pattern is not only an architecture. One of the most useful aspects of patterns is that they exhibit known quality attributes. Some patterns are chosen for performance problems, some are for high-security problems and some are for high-availability. They are also called architectural styles.

A pattern is made up of three related parts:

1. Context. Design situation giving rise to a design problem.

2. Problem. Set of forces repeatedly arising in the context.

3. Solution. Configuration to balance and release the forces, including: Structure of components and relationships, and Run-time behaviour.

Here the Context is the running environment for a software system, which will suggest the architecture pattern as its basis for architecture design.

The problems are what we are going to face when we design a software architecture. They are composed of a set of forces, which could be grouped into two classes, one of which is functional forces and the other is

quality attribute forces. All aspects of the problem must be solved using the pattern. Of course, there may be lots of special problems in a concrete application and a pattern cannot solve all kinds of such special problems. It only matches the common problems which it is designed to solve, such as fault tolerance, voting redundancy, load balancing and so on.

The solution is presented as a structure, most often as a logical structure which can solve the problems, most of which are quality problems. This is the main part of a pattern, and we will first think of the structure when we mention a pattern.

Different patterns have different benefits and liabilities and can solve different problems,

Shaw and Garlan first published a collection of architecture patterns in [31]. The collection includes:

- Independent components: communicating processes, implicit invocation, explicit invocation;
- Data-centric: repository, blackboard;
- Data flow: batch sequential, pipe and filter;
- Machine: interpreter, rule-based system;
- Layer: main program and subroutine, object-oriented, layered.

Buschmann and others [4] entended the collection, adding some new patterns, like MVC (Model View Controller), Fault Tolerant, Parallel pipeline and so on. Buschmanns categorized the patterns into the following four new groups:

- From Mud to Structure. Patterns in this category help you to avoid a 'sea' of components or objects. In particular, they support a controlled decomposition of an overall system task into cooperating subtasks. The category includes the Layers pattern, the Pipes and Filters pattern, and the Blackboard pattern.
- Distributed Systems. This category includes one pattern, the Broker, and refers to two patterns in other categories, Microkernel and Pipes and Filters. The Broker pattern provides a complete infrastructure for distributed applications. Its underlying architecture is soon to be standardized by the Object Management Group (OMG). The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories.

- Interactive Systems. This category comprises two patterns, the Model-View-Controller pattern , well-known from Smalltalk, and the Presentation -Abstraction -Control pattern. Both patterns support the structuring of software systems that feature human-computer interaction.

- Adaptable Systems. The Reflection pattern and the Microkernel pattern strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Note that this categorization is not intended to be exhaustive. It works for the architectural patterns described in [4,18], but it may become necessary to define new categories if more architectural patterns are added.

1.3.2 Relations of Different Patterns

A close look at many patterns reveals that, despite initial impressions, their components and relationships are not always as 'atomic' as they first appear to be. A pattern solves a particular problem, but its application may raise new problems. Some of these can be solved by other patterns. Single components or relationships inside a particular pattern may therefore be described by smaller patterns, all of them integrated by the larger pattern in which they are contained.

Most patterns for software architecture raise problems, some of which can be solved by smaller patterns. Patterns do not usually exist in isolation.

Each pattern depends on the smaller patterns it contains and on the larger patterns in which it is contained. This means all the patterns in different levels of a hierarchy of patterns are dependent on each other. A pattern may also be a variant of another. From a general perspective, a pattern and its variants describe different or similar solutions to very similar problems. These problems usually vary in some of the forces involved, rather than in general character.

Patterns can also combine in more complex architecture structures at the same level of abstraction. This happens when the original problem includes more forces than can be balanced and solved by a single pattern. In this case, applying several patterns and combining them in some way can solve the problems, and each such smaller pattern can resolve a particular subset of the family of forces.

All three kinds of relationship refinement, combination, and variants help in using patterns effectively. Refinement supports the implementation of a pattern, combination helps you compose complex design structures,

and variants help when selecting the right pattern in a given design situation (design environment and problem).

Now, we know all the patterns can be chosen singly to solve some concrete problems or can be chosen to solve a set of problems cooperatively.

1.3.3 Architecture Pattern and Quality Attributes

According to the definition of architecture pattern, it is very clear that one pattern is used to achieve one or more quality attributes. And most of the time, more than one pattern can achieve the same quality attribute, and one pattern can have the properties for more than one quality attribute.

Using a pattern to realize the quality attribute requirement is now very popular in practice, because it is reliable and economical. The patterns are repeatable for the same or similar problems, but most of the time, we are going to solve the problem for more than one quality attribute, that will need more than one pattern to work together and get the best balance tradeoff, which we can relate to the relation of combination of different patterns described above.

1.3.4 Architecture and Architecture Pattern

In designing a software architecture, as we describe above, the main objective is to achieve the quality attribute requirement. Each architecture pattern is designed for one or more quality attribute strategies. We define a quality attribute strategy as a method or tactic used to achieve the quality requirement during system design and development.

For each quality, there are identifiable strategies (and patterns that implement these strategies) that can be used in an architecture design to achieve a specific quality. Each strategy is designed to achieve one or more quality attributes, but the patterns in which they are embedded may have an impact on other quality attributes. In an architecture design, a composition of many such strategies is used to achieve a balance between the required multiple qualities. Achievement of the quality and functional requirements is analyzed during the refinement step. The availability quality is the main focus for this thesis, and of course, there are several special patterns designed for implementing the strategies for high availability; but our emphasis will be on how to improve the availability of an entire architecture of a system, composed of different patterns, which may not be designed particularly for high availability.

But, a single pattern cannot enable the detailed construction of a complete software architecture, while it just helps us to design one aspect

of our application. Even if we design one aspect correctly, the whole architecture may still fail to meet its desired overall quality requirements. To meet the needs of software architecture as much as possible, we need a rich set of patterns that must cover many different design problems. The more patterns that are available, the more design problems that can be addressed appropriately, and the more we are supported in constructing software architectures with defined properties. On the other hand, the more patterns that are available, the harder it is to achieve an overview of them. As we have already pointed out, there are many relationships between patterns. When applying one pattern, we want to know which other patterns can help refine the structure it introduces. We also want to know which other patterns we can combine with it and which other variant patterns we can choose to achieve the same property and some other properties at the same time.

Of course choosing the pattern is one part of software architecture design, and there are lots of other jobs to do for completing the architecture design, such as the task of integrating the application's functionality with the framework, and detailing its components and relationships, perhaps with help of design patterns and idioms. The selection of an architectural pattern, or a combination of several, is only the first step, but the most important step, when designing the architecture of a software system.

1.3.5 Architecture Decision and Availability

We notice that redundancy is now becoming popular for Server farms, and redundancy is the most sensitive factor that can improve availability. We will discuss this in the Chapter on Availability Strategies. In addition, load balancing is a kind of special strategy, which can improve performance quality and availability at the same time.

Both of these strategies are good strategies for improving availability, and many others will be introduced in Chapter 4. But how an architecture strategy decision may influence the quality attributes, or availability attribute is a problem we will discuss in this thesis first. The relationship among Context, Architecture Patterns, Architecture decisions and High Availability strategies can be illustrated below in figure 2

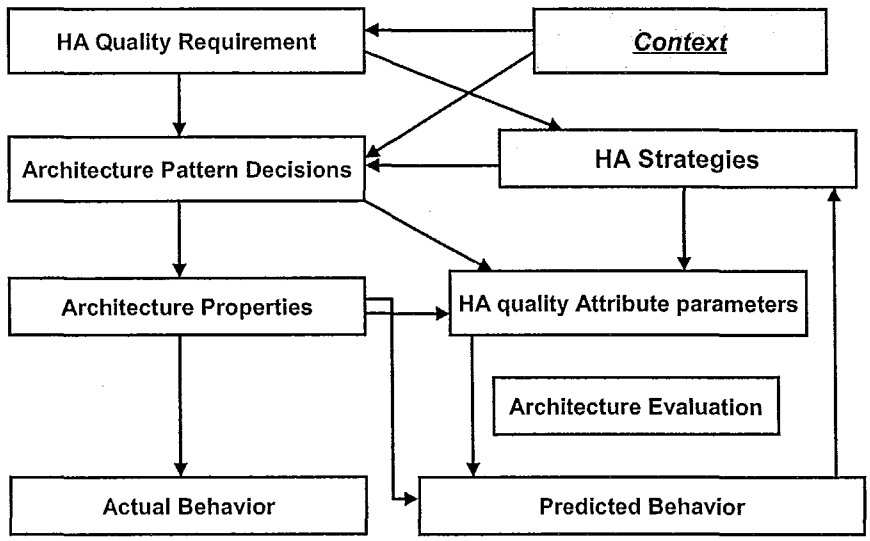

Figure 2: Architecture Design Decision and Predicted Behaviour

This figure illustrates the relationship between the context and architecture choice and, at the same time, describes the relationship among quality requirements and architecture (pattern) decisions, the quality strategies, quality property parameters, and the evaluation of the architecture structure.

The HA requirement is based on some context, and it will decide the architecture decision and HA strategies decision in addition to the context. The architecture pattern decision will decide the architecture properties, which will decide the HA quality attribute parameters, like the redundancy policy, take-over policy, detect time and roll-over time. Based on the architecture properties, including the logical structure of the architecture,

and the HA parameters, like individual availability of a component, we can evaluate the availability of a system by using the PBSA model . By comparing the result of evaluation with the HA requirement, we can decide on the more strategies to take or the completion of an architecture design for high availability.

We will discuss how to predict or analyze the availability from architecture structure, based on pattern decisions. How to make the pattern choice is not what we want to discuss here, but the availability strategy decision will be discussed later.

2 Qualitative Availability Analysis

2.1 Why do we need Quality Analysis?

As we described in Chapter 1, the main purpose of architecture design is to make some decision about how to separate functional properties, how to choose architecture patterns and how to realize the whole requirements for functional and qualitative needs.

But, the choices of architecture patterns are based on the concrete quality requirements, which may be defined by customers or by architects, according to different scenarios.

During the requirement definition or specification, the execution environment or context is very important, because it sometimes can decide whether the requirement is achievable or not before actual architecture design, for example a six 9 availability requirement using only one server.

After making the architecture pattern decision, and getting an original architecture structure, how do we know whether the architecture design can fit our requirement or not, especially our quality requirement? As we know, there is no actual mechanism to decide whether the quality attribute of an architecture fits for our requirement. This is because, firstly, the requirements for quality attributes are not quantitatively described in most practical projects and, secondly, our predictive model for these qualities are not quantitatively defined.

So, until now, most of the time architects have to use a scenario-based model [10], to do availability qualitative analysis. Also, Mark Klein [17] introduced a Markov model based method for qualitatively analyzing. Neither of the models can really answer a question like: "What is the exact availability of the system?". Someone may answer this question as some nines for the servers, but call that the servers are only some of the components of a system. It cannot represent the entire system. In addition, without a quantitative model, we can not know exactly how to improve the availability and we don't know how to choose the most sensitive component, and maybe we spend much money on the system, but the reward in terms of the quality still remains not as good as we expect.

So, we need a model for quantitative analysis. And in this Chapter, we will first introduce a qualitative model, introduced by Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson first [17], and we will see how it works, then we will introduce another qualitative method for quality analysis. And we will introduce a quantitative model of availability in the next Chapter.

2.2 Scenario-Based Quality Analysis

2.2.1 What is Scenario-Based Quality Analysis?

Rick Kazman, Gregory Abowd, Len Bass, Paul Clements defined a Scenario-Based method for quality analysis in [10]. Scenarios are important tools for exercising an architecture in order to gain information about a system's fitness with respect to a set of desired quality attributes. In the thesis [10], Kazman and others presented a set of experiential case studies illustrating the methodological use of scenarios to gain architecture-level understanding and predictive insight into large scale, real-world systems of various domains and a structured method for scenario-based architectural analysis is presented, using scenarios to analyze architectures with respect to achieving quality attributes.

Scenarios have been widely used and documented as a technique during requirements refinement and elicitation, especially with respect to the user and maintainer of the system [10]. They have also been widely used during design as a method of comparing design alternatives. Experience also shows that programmers use them to understand an already-built system, by asking how the system responds (component by component) to a particular input or operational situation. Before being presented by Kazman [10] for quality analysis, scenarios had already been well used as a tool in the requirements stage and in the design and programming stages.

Kazman and others used Scenarios as a tool for analysis of quality in this thesis. They use scenarios to express the particular instances of each quality attribute important to the customer of a system. Then they analyze the architecture under consideration with respect to how well or how easily it satisfies the constraints imposed by each scenario.

Scenarios used in the analysis of quality should include all the roles involved in the system, from operator to manager and maintainer. And also Scenarios, for the future use of the system, should be considered ; this property leads to the fact that not all the scenarios can be listed at the architecture design stage.

Recall that a scenario is a brief description of some anticipated or desired use of a system [10]. It may be the case that the system directly supports that scenario, meaning that the anticipated use requires no modification to the system in order to be performed. This would usually be determined by demonstrating how the existing architecture would behave in performing the scenario. If a scenario is not directly supported, this means that there must be some change to the system that we could represent architecturally.

2.2.2 A Method of Scenario-based Analysis

A particular method for doing a scenario-based architectural analysis is SAAM (Software Architecture Analysis Method). SAAM was originally developed to enable comparison of competing architectural solutions [32]. Not all of the experience of architectural analysis has strictly followed the method prescribed by SAAM, and it has not always been the case that we were comparing competing candidate architectures. Nevertheless, in all cases, scenarios were used as the foundation method when qualitatively illuminating the quality properties of an architecture, and from this body of experience, we get a set of stable activities and their dependencies, which organized a method SAAM, and we can use it appropriately.

SAAM includes these steps:

1. **Describe candidate architecture.** The candidate architecture or architectures should be described in a syntactic architectural notation that is well-understood by different parties involved in the architecture analysis. These architectural descriptions need to indicate the system's functional computation properties and data components, as well as all component relationships, which we call connectors. There are many such syntactic architecture description languages we can use, i.e. Wright, Acme and so on.

2. **Develop scenarios.** Develop task scenarios that illustrate those activities the system must support and those changes that it is anticipated may be made to the system when using it or in the future. In developing these scenarios, it is important to consider all important roles of a system. Thus the scenarios designed will represent tasks relevant to different roles such as: end user/customer, marketing, system administrator, maintainer, and developer.

3. **Perform scenario evaluations.** For each indirect task scenario (which means that the current system does not support it, and some changes should be made to support it), list the changes to the architecture that are necessary for it to support the scenario and estimate the cost of performing the change. A modification to the architecture means that either a new component or connection is introduced or an existing component or connection requires a change in its specification. By the end of this stage, there should be a summary table which lists all scenarios (direct and indirect). For each indirect scenario the impact, or set of changes, that scenario has on the architecture should be described. A tabular summary is especially useful when comparing alternative architectural candidates because it provides an easy

way to determine which architecture better supports a collection of designed scenarios.

4. **Reveal scenario interaction**. Different indirect scenarios may need changes to the same components or connections. In this case we say that the scenarios interact in that component or connector. So, in some point, determining scenario interaction is a helpful process for identifying scenarios that affect a common set of components or connectors. SAAM favors the architecture with the fewest scenario conflicts.

5. **Overall evaluation**. Finally, weight each scenario and the scenario interactions in terms of their relative importance and use that weighting to determine an overall ranking. This is a subjective process, involving all of the stake-holders in the system. The weighting chosen will reflect the relative importance of the quality factors that the scenarios manifest. In addition, the ranking can help discover the most important concern of a system, whether it be availability or performance or something else.

Note: there may be several method or views to represent an architecture of a system, as we discussed before, but in the SAAM method, we only need a simple one, maybe a functional or logical view to represent the architecture. Of course, it will be useful to choose the most suitable one for your own company or organization, as long as most of the analysis team members can understand what the architecture represents.

2.2.3 Analysis and Conclusion

This method is a very useful method for us to evaluate an architecture design, especially to help for us to choose between different competing candidates.

But as we know, the scenarios are different and vary according to different users and roles. And different stake-holders may have different concerns, and the quality requirement is very difficult to elucidate. Even the users with the same responsibility and same role will have different concerns and can give different scenarios according to their different experience. So, clearly, it is very hard to evaluate a system design very precisely using the SAAM.

The reason why we say it is hard to evaluate a software architecture design is because the method is a qualitative method; different analysts can come to different results. So we need some quantitative model with absolute evaluation data for quality analysis.

2.3 Another Qualitative Model

In [17], Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, and Mario Barbacci introduced a qualitative model for availability.

They first introduced a pattern for some problem and, by improving the pattern, they hope to get better availability. They use a Markov model to qualitatively describe this improvement.

2.3.1 A Simplex Pattern

A pattern, named simplex, is introduced in [17], which belongs to a general family of Availability patterns that could be called redundancy patterns. The general pattern for a redundancy pattern is shown in Figure 3 below.

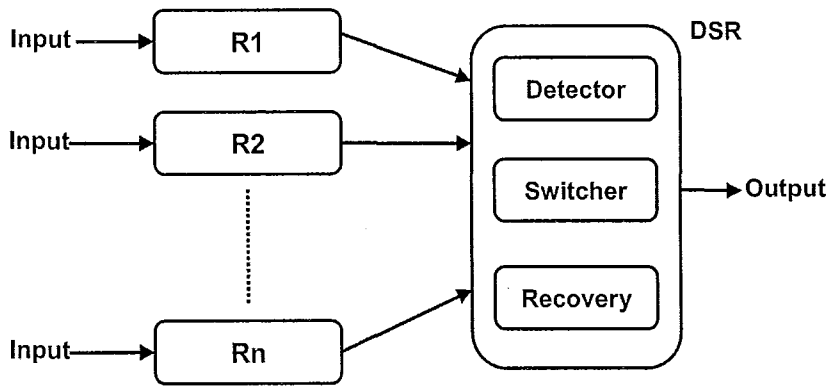

Figure 3: A Redundancy Specific Architecture Pattern

The pattern, from an availability point of view, consists of multiple redundant components. Data flows into one or more redundant components, which then send their output to another component (or possibly components), which is (are) responsible for detecting failures, switching to a working component and possibly initiating recovery of the failed component, according to the situation, whether we find a failure or fault[2].

This is a pattern for the quality of availability, which is used to increase the availability of component R_1, the "leader" of the redundancy components. The outputs of all the redundant components are input into the component DSR, which will compute an algorithm, using all the outputs as its inputs to choose a correct one. If the algorithm runs with a failure output from R_n, it will call a process to switch the lead component

[2]We will not differentiate fault and failure in this thesis.

R_n to R_{n+1}, and call a recovery process to make R_n recover and let it be a new back up component.

The Simplex pattern, as shown in Figure 4, is a variant of the redundancy pattern, in which the redundant components are processes. The components do not necessarily receive the same input or generate the same output. Moreover, the components are not all peers. The components are *analytically redundant*, meaning they are redundant with respect to the general effect their output has in controlling their environment, but not necessarily redundant in the algorithms used or the output produced[3]. The "leader" component, the other redundant components (R_1 and R_2) and the "safety" component are analytically redundant.

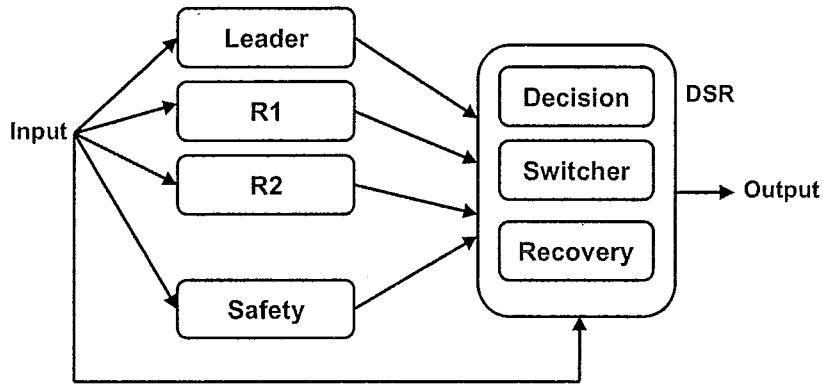

Figure 4: Simplex Architecture Pattern

The "leader" is typically the upgraded version of a critical component, which we will call the most sensitive component. All components execute concurrently. The leader's output is used if it passes the acceptance test applied by the DSR unit. The acceptance test is based on a model of the controlled environment and the ability of the safety component to recover from actions of the other components. If the leader does not pass this test, a new leader is picked (either R1 or R2). The "safety" component is a simple, highly available analytically redundant component that is used as a last resort. The safety might be used to affect a recovery to the point where one of the other (more able) components can once again take over. Note that the DSR component receives a copy of the input and uses it as a basis for performing its acceptance test. The Simplex pattern assumes that mechanisms exist to bound the execution time of the components, thereby preventing timing overruns. Another (related) pattern would address performance issues. The Simplex pattern also assumes that the concurrent

[3]This is just like the relationship between the *ABS* and a foot brake; although they do the same braking work, the procedure, the performance and the result may be different.

units are processes with address space protection, thereby preventing the propagation of system faults such as memory overruns.

The main improvement of this pattern from the original redundancy pattern is that it need not choose which one is the correct one, and it can quickly recover from one leader component's failure, because of concurrency. That will reduce the time of recovery, and increase the availability for this component R_1 or R_2.

2.3.2 Modeling the Simplex Pattern

Let $n = 3$ for the pattern, illustrated in Figure 3. The Voting algorithm needs at least 2 or all 3 components producing results that agree, otherwise the system has failed. Let us look at all the states the system can be in and offer normal service. The system can be in one of three states: it has 3 working components; or it has 2 working components; or it failed. If there are F failures per year and a component failure repair takes on the average 1/R years, then the Markov model shown in Figure 5 can be used to show the quality of availability (that is, the proportion of time that the system is not in the failed state).

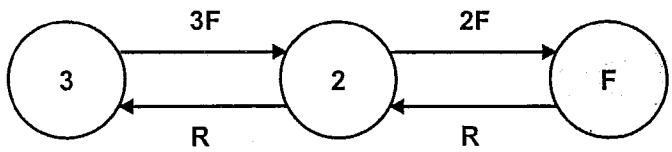

Figure 5: A Markov Model for Majority Voting

The representation of a Markov model in Figure 5 can be viewed as a state diagram. State "3" represents the state in which 3 components are available, state "2" represents the state in which 2 components are available and the grey state "F" is the failed state. The transition arrows between these states are labelled with failure (F) and repair (R) rates. Since each component fails independently with an average rate of F, 3 components fail with an average fail rate of $3F$ and hence the label for the transition from state "3" to state "2" is $3F$.

The steady state solution of the Markov model yields the long-term proportion of time that the system is in each state. Therefore the availability of the majority voting case is the proportion of time in which the system is in state "3" or state "2", and hence not in the failure state "F".

This is only a model for qualitatively illustrating the availability of a system component, not for the entire system. It has not give us the absolute value of availability, and we will see how it is used to show the

improvement of availability.

Modeling the Simplex Pattern

The Simplex pattern achieves a relatively high level of availability of the high performance (e.g., a very precise algorithm) variant by using a highly available but lower performing (e.g., a less accurate algorithm) variant to recover from faults. To illustrate the concept consider a system with two redundant components (R1 and R2), a safety component, and a monitoring, decision and recovery unit. The Simplex pattern preserves the total number of active components, but allocates functions to components differently depending on their states, and hence the components have different failure properties. The Markov model for this style is shown in Figure 6.

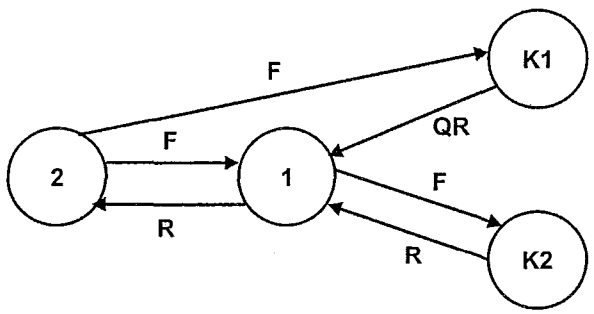

Figure 6: A Markov Model for Simplex

The system starts in state "2" with two functional high performance components, the outputs of which will be compared. If they agree, we assume that they are correct; this means we assume no common mode failure[4], but rather random failures. If they disagree, one is picked. If the right one is picked, the model transfers from state "2" to state "1". If the wrong one is picked the model transfers from state "2" to state "K1", where K1 stands for a state in which the safety component becomes active. Since one of the high performance functional components continues to work, the transition from "K1" to "1" is relatively quick and thus has a quick repair (QR) rate. We assume that $QR = n \times R$, for some $n \geqslant 1$. If a failure occurs while the system situation is in state "1", the system also transfers to the safety component, but in this case the repair rate is that of a normal repair (i.e. a software or hardware fix).

Of course, the algorithm is trying to guarantee that there is a component available, while every time it transfers the system state to a state,

[4]This assumption will be throughout this thesis, so, in this thesis, no common mode failure will be considered

in which the safety component is activated. If both the functional components have failed, the system will be in failure state "K2", even if the safety component is activated. So, each time, when the safety component is activated, the controller of recovery component will call for recovery work for the failed component at the same time.

A key to the availability properties of this pattern is the relatively quick repair rate (QR) from state "K1" to state "1". To see this, imagine that QR is so quick that virtually no time is spent in state "K1". In this case the model in Figure 6 closely approximates the model in Figure 7, below. The availability properties of the model shown in Figure 7 are better than for majority voting (shown in Figure 5) due to the higher transition rates for majority voting. The higher transition rates for majority voting are a consequence of needing a majority of the redundant components to agree in order to detect a failure, whereas this pattern uses a semantic check of the output for failure detection.

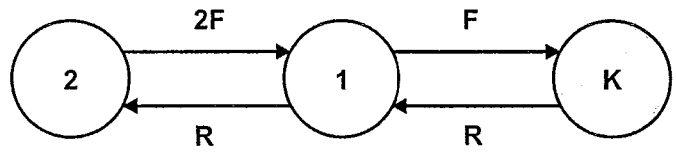

Figure 7: An Approximate Markov Model for Simplex

So, the main difference between these two variants of the redundancy pattern is in the algorithms, which the decision component uses to decide which one is the correct answer for output. The algorithm for the Majority Voting pattern is based on the agreement of more than 1 component, and the algorithm for Simplex is to choose the output semantically. The difference in the performance of these two algorithms implies the different performance of the redundancy patterns, and the difference in the availability property.

2.3.3 Analysis and Conclusion

In the section above, we introduced two variants of the redundancy pattern, used by Marks for their *attribute-based architecture pattern.* These two patterns are redundancy patterns for the availability attribute. They clearly show that availability is improved by using these patterns and they also show that the Simplex pattern is better than the Majority Voting one in performance and availability, by using Markov Model.

We now know these two patterns can be used to improve the availability of a key component. But the problem is still there, that is how

to choose the critical component, and how to demonstrate that the entire system, including more than one architecture pattern, has an improved availability property after using such a redundancy pattern?

Mario R. Barbacci and some others improved this Markov Model for redundancy architecture pattern [35], and gave some algorithms for calculating the availability of Servers.

To conduct the availability analysis, they assign each component a failure rate and a repair rate, the rate at which this component recovers from a failure, obtained from questionnaires or checklists, depending on the maturity of the domain or prior experience with similar components. To understand the availability of the RTS, they use a machine repair model with S machines and one repairman. The amount of time each machine operates before breaking down is exponentially distributed with mean $1/\lambda$ (the failure rate of machines is λ). The amount of time that it takes to repair is exponentially distributed with mean $1/\mu$ (the repair rate is μ). Both λ and μ are discovered parameters needed by the availability model.

In the machine repair model [35], they say that the system is in state n whenever n machines are not it use. They say that the system is available (albeit with diminished capacity) whenever at least one server is operating (available) and they say that the system is down whenever all S servers are down. The long-run proportion of time that the system is not in state S (i.e., the system is available, by their definition) is given by

$$A = 1 - \frac{S! \cdot (\frac{\lambda}{\mu})^S}{1 + \sum_{n=1}^{S} (\frac{\lambda}{\mu})^n \times \frac{S!}{(S-n)!}} \tag{2}$$

But we can also see this as a special pattern for component redundancy, because they only consider the availability of redundant servers, which in an architecture structure may be only a component or a pattern, but there remains the problem of how to calculate the entire system with many patterns and how to choose the critical components(s) for the redundancy pattern? This is what we want to introduce next.

In summary, we now know that there are at least two models for quality analysis, one of which is Scenario-based quality analysis model; the other one is a model based analysis method. Both of them are qualitative quality attribute analysis methods, and they have the following drawbacks:

1. No accurate assessment for availability;
2. No hint for improving availability;
3. Difficult for complex architecture.

2.4 Fault Tree Analysis

In the technique known as fault tree analysis (FTA), an undesired effect is taken as the root ('top event') of a tree of logical expressions. Then, each situation that could cause that effect is added to the tree as a node characterizing that cause, in terms of a logic expression. When fault tree nodes are labeled with actual numbers about failure probabilities, which are often in practice unavailable because of the expense of testing, computer programmers can calculate failure probabilities from fault trees.

Sometimes, this is a useful method for Reliability and Safety calculation and prediction for some computer systems. The Tree is usually written out using conventional logic gate symbols. The route through a Tree between an event and an initiator in the tree is called a Cutset. The shortest credible way through the tree from Fault to initiating Event is called a Minimal Cutset. For different systems, we can analyze and obtain all their Fault Trees and, according to the testing result, we can get the probabilities of all their initiators and events, even though it will be very hard work.

The prediction and calculation of Reliability and Safety of some computer systems using Fault Tree Analysis are based on these probabilities of initiators and fault events. Using this technique, we can calculate the failure probability of a computer system, if we can create all failure trees and get all the probabilities for the initiators and events. But for calculating availability, we still need to know the information about repair time. According to the definition of availability, and the difference between availability and reliability, we cannot calculate availability using Fault Tree Analysis technology currently, because of the lack of such repair time information from FTA.

In summary, Fault Tree Analysis is a good and feasible technology for Safety and Reliability calculation and prediction, if we can get all the probabilities of initiators and events. But, we cannot calculate and predict availability using FTA, because we cannot get information about repair time from it. So we have defined the PBSA model for availability calculation and prediction.

3 Quantitative Availability Analysis

3.1 Simple Part Availability Calculation

To model how to calculate the availability of a complex system with complex structure and multiple patterns such as the one shown in Figure 1, the Business Intelligence system, we should first look at how a simple part is calculated.

The availability of a simple system or basic part of a system is calculated by modeling the system part as an interconnection of components in series and parallel. The following rules are used to decide if components should be placed in series or parallel:

- If failure of a component leads to the combination becoming inoperable, the two components are considered to be operating in series
- If failure of a component leads to the other component taking over the operations of the failed one, the two components are considered to be operating in parallel.

So, we define a *part* as:

Definition 3.1 *A* **Part** *of a system is a subset P of a set of components A, which can be viewed as a new component C, with the same availability relation with the other components outside of P in A as the availability relation between P and A, that is $R(C, A-P) = R(P, A-P)$.*

In Figure 8, both of (a) and (b) are parts of an architecture structure. (a) is a series part and (b) is a parallel part.

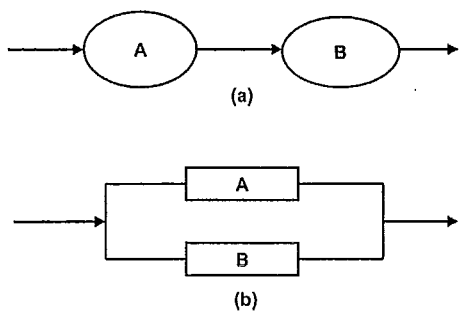

Figure 8: Parts of Architecture Structure

But Figure 9 shows a structure, which cannot be viewed simply as a part of such an architecture structure, and we can call it a **Non-Part**,

because we cannot reduce it to a single part, using current rules (will be discussed later).

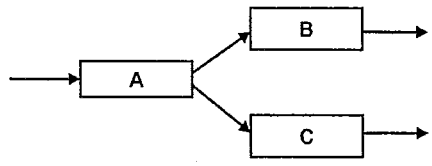

Figure 9: An Example of Non-Part

We call this a non-part because firstly, if one of the two components B and C fails, the system will still be available on the other path, so it is not series; secondly, component B(C) will not take the role of C(B) when C(B) has failed.

And we can view one component as a part too, so we have:

Axiom 3.1 *A component in an architecture structure can be a* **Part**.

3.1.1 Availability in Series

As stated above, in Figure 8(a), two components A and B are considered to be operating in series, if failure of either of the components results in failure of the combination. The combined system is available only if both component A and component B are available. From this, it follows that the combined availability is a product of the availability of the two components, just like the reliability calculation for this structure in [8]. The combined availability is shown by the equation below:

$$A = A_A \times A_B \tag{3}$$

where A_A and A_B are the availability of components A and B respectively. We assume that the availability of a single component, like A_A is known, according to the experience of existing systems and technical reports from vendors of different products, like a server.

The implications of the above equation are that the combined availability of a part in series is always lower than the availability of its individual components. Because $A_A \leqslant 1$ and $A_B \leqslant 1$, so $A \leqslant 1$. Consider the system in Figure 8(a) above, where component A and B are connected in series. The table below shows the availability and downtime for individual components and the series part.

Component/Part	Availability	Downtime
A	99.9%	8.76 hours/year
B	99.99%	52 minutes/year
Part(a)	99.89%	9.64 hours/year

The above table clearly shows that even though a very high availability component B was used, the overall availability of the system was pulled down by the low availability of component A. This just proves the saying that a chain is weaker than the weakest link.

3.1.2 Availability in Parallel

As stated above in Figure 8(b), two components are considered to be operating in parallel if the part is considered failed when both parts fail. The combined system is available if either of them is available, in other words, only one of them failed. It follows that the availability of the parallel part is 1 - $P_r\{both\ components\ are\ unavailable\}$, as in the reliability evaluation in [8] for the same structure; the difference is that the latter one uses the reliability of those components, but we use availability. The parallel part availability is given by the equation below:

$$A = 1 - (1 - A_A) \times (1 - A_B) \tag{4}$$

The implication of the equation above are that the parallel part availability of two components in parallel is always much higher than the availability of its individual components. Consider the system in Figure 8(b). Two instances of component A and B are connected in parallel. The table below shows the availability and downtime for individual components and the parallel combination.

Component/Part	Availability	Downtime
A	99.9%	8.76 hours/year
B	99.9%	8.76 hours/year
Part(b)	99.9999%	31.54 seconds/year

The table above clearly shows that, even though a very low availability component A was used, the overall availability of the system is much higher. Thus parallel operation provides a very powerful mechanism for making a highly available system from low availability components.

This is a simple pattern of redundancy. If we add more components in the parallel part, the overall availability will become even stronger. We will introduce the detailed way to calculate the availability of this pattern.

Note: the meaning of *parallel* here is different from the usual parallel pattern. In a parallel pattern, the availability of the parallel pattern part is defined as $P_r\{$ *All the parallel components are available* $\}$. Actually, *parallel* for a part is very much like a simple redundancy pattern, which we will introduce later.

Availability in a Combinatorial Part

Consider the problem shown in Figure 10. Components (B, B') and (C, C') are working in parallel, meanings that if either of them fails, the other one will take over its role. Components B and B' are working in series, components C and C' work in series too.

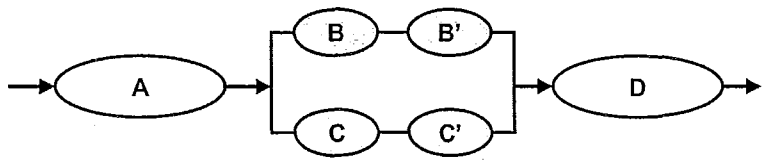

Figure 10: An Example of Combinatorial Parts

Now, we want to calculate the overall availability of this system. We consider this system as a system with 4 parts, three of them are series parts and one parallel part. We illustrate it as in Figure 11 below.

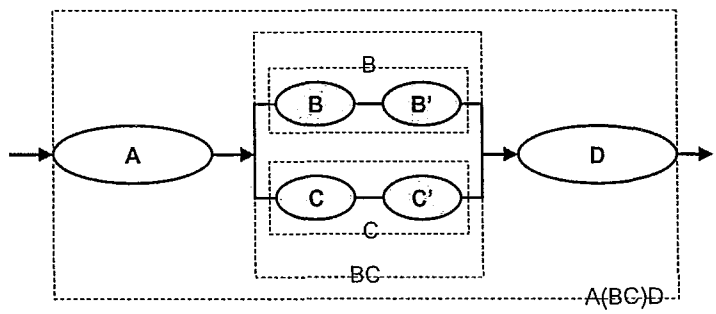

Figure 11: A Part View For Figure 10

Suppose we already know the availability for all the components A, B, B', C, C',and

D, and they are listed in the table below, then we can calculate the availability of different parts as follows,

Component/Part	Calculation	Availability	Downtime(hour)
A	A_A	99.9%	8.76 /year
B	A_B	99.9%	8.76 /year
B'	$A_{B'}$	99.9%	8.76 /year
C	A_C	99.9%	8.76 /year
C'	$A_{C'}$	99.9%	8.76 /year
D	A_D	99.9%	8.76 /year
Part(B)	$A_B \times A_{B'}$	99.8%	17.52 /year
Part(C)	$A_C \times A_{C'}$	99.8%	17.52 /year
Part(BC)	$1-(1-A_{P(B)}) \times (1 - A_{P(C)})$	99.9996%	3.504 /year
System($A(BC)D$)	$A_A \times A_{P(BC)} \times A_D$	99.7997%	17.5463 /year

Without considering common mode failure, the availability of a simple combinatorial part is easy to calculate by looking at the lower level part as a new component, and extending the definition of part into a new part which can be made up of smaller parts, which are in series or in parallel.

In the example above, the part(BC) is made up of two smaller parts: part(B) and part (C); they are working in parallel, so part(BC) is a new parallel part, made up of 2 series parts. And the system can be looked at as a big series part, which is made up of three components, component A, component BC, and component D. Component BC is the part (BC), which can also be viewed as a component.

So, we have the following definition and axiom.

Definition 3.2 *A* **Superpart** *is a part, which is made up of more than one smaller parts in series or parallel.*

Axiom 3.2 *A part can be viewed as a component, and a component can be viewed as a part in a logical architecture view.*

Axiom 3.3 *A* **transition** *from a component-based view architecture structure to a part-based view of an architecture structure will not change the functional property and the availability property of the system.*

Now, that we know how to calculate the availability of a simple combinatorial part, we will introduce some more complex pattern-based combinatorial parts below.

Can we claim that the availability of each system will be calculable using this simple part calculation method? Let us look at an example below in Figure 12.

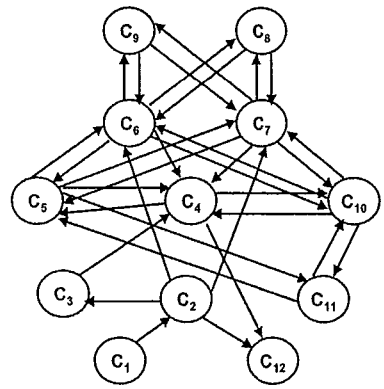

Figure 12: A Complex Architecture Structure

We can see from above Figure 12, components C_8 and C_9 are working in parallel, and component C_6 and C_7 are working in parallel too. We can change this component-based view of an architecture structure into a part-based view of an architecture structure, as in Figure 13 below, by reducing some of the patterns into single parts, according to AEAR (Architecture Equipollence Availability Reduction), which will be introduced later.

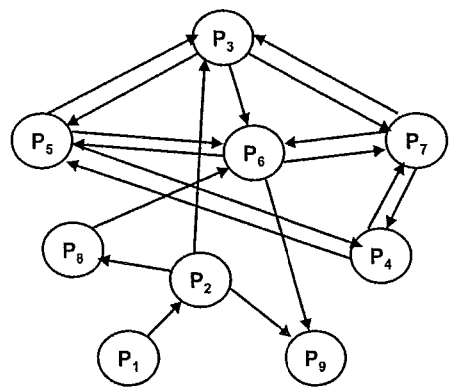

Figure 13: A Part-Based Architecture View

Obviously, this system still includes too complex a structure for our simple part calculation method to calculate the availability of the overall system. Many components or parts have more complex relationships than simple series or parallel as described above. So, we would like to introduce a new model, based on the simple part calculation idea and Markov Transition systems, to predict and calculate the overall system availability for such a large scale complex system.

3.2 A PBSA Model

3.2.1 Markov Chain

Before we introduce the PBSA Model, we would like to introduce some ideas about Markov Chains, which is the foundation of PBSA.

In mathematics, a Markov chain, named after Andrey Markov, is a discrete-time stochastic process with the Markov property.

A Markov chain describes the states of a system at successive times. At these times the system may have changed from the state it was in the moment before to another state, or it may have stayed in the same state. The changes of state are called transitions.

The **Markov property** means that the conditional probability distribution of the state in the future, given the state of the process currently and in the past, depends only on its current state and not on its state in the past. This has three meanings:

1. the state transition from state 1 to state 2 only depends on the current state 1, not the history of the system;
2. the state transition from state 1 to state 2 will not depend on other states;
3. each transition is a discrete-time stochastic process. It may be continuous, but in this thesis we may think it is discrete.

So, we have the definition of a Markov Chain, as in [33]:

Definition 3.3 *A Markov chain is a sequence of random variables* $X_1, X_2, X_3, \cdots$ *with the Markov property, namely that, given the present state, the future and past states are independent. Formally,* $P_r[X_{n+1} = x | X_n = x_n, \cdots, X_1 = x_1, X_0 = x_0] = P_r[X_{n+1} = x | X_n = x_n]$

The possible values of X_i from a countable set S are called the state space of the chain.

Here are some important properties of a Markov Chain:

- Define the probability of going from state i to state j in n time steps as, $p_{ij}^n = P_r[X_n = j | X_0 = i]$
- The single-step transition as, $p_{ij} = P_r[X_1 = j | X_0 = i]$.
- The n-step transition satisfies the Chapman-Kolmogorov equation, that for any $0 < k < n$, $p_{ij}^n = \sum_{r \in S} p_{ir}^k p_{rj}^{(n-k)}$

- The marginal distribution $P_r(X_n = x)$ is the distribution over states at time n. The initial distribution is $P_r(X_0 = x)$. The evolution of the process through one time step is described by, $P_r(X_{n+1} = j) = \sum_{r \in S} p_{rj} P_r[X_n = r] = \sum_{r \in S} p_{rj}^n P_r[X_0 = r]$
- The superscript (n) is intended to be an integer-valued label only in this thesis;

A Markov Chain corresponds to a transition matrix; an element of the matrix is the probability of transition from one state to the other, and the matrix should satisfy the property above.

An example of using a Markov model is introduced by Cheung, for his model of reliability in [5], and another example of using a Markov Chain is introduced by Clegg, for his Internet Traffic Model in [34]. We will also use the properties of a Markov Chain and the Transition Matrix as a foundation during the building of our part-based availability model.

3.2.2 PBSA Model

Now, we will introduce a quantitative PBSA (part-based software model for availability), which can be used to calculate the entire availability of a complex system, where we assume that all the components are simplified or reduced into *pure parts.*

Definition 3.4 ***A pure part is a part, which can functionally work independently, which means it is available independently.***

Because the parts in the structure are pure parts, we can consider them as different independent states in a Markov chain. We first look at the structure as a directed graph, each node of the graph is a part, and an edge of the graph represents the relation between one part and the other; this relationship may be the sequencing of different functions, or the data flow among different parts. We do not care about what kind of relationships they are, and we just think of the relationship as a transition from a state to another. And the availability of the second part will not depend on the previous one, and we will see how to make sure of this in the next Chapter.

The independence here has two different meanings, one of which is that the independence of a part means that the part can execute a function independently, and its failure will not cause the other part to be unavailable. For example, in a C/S pattern system, a failure of a client part will not compensate to make the server fail too. And even if there is such a possibility, for example a virus, we assume that they are independent and are not compensating. The second meaning of independence is that,

if a part is unavailable, it cannot be taken over by an other part, as in a redundant pattern. So, we say in our model, if a part is unavailable, the total system will be unavailable at this moment. Remember, here the part is an independent part, and its availability will not be affected by any other parts, and a redundancy pattern should be reduced into a pure part before we use PBSA.

Because a part is composed of some components, and we assume in this model that the availability of a component is defined and will remain invariant, until something updated for it. We will discuss how to improve the availability of a component in the next Chapter. Here, we first assume that we already know the availability of each component, and so each part. The calculation of the availability of a part is discussed in the previous section and we will discuss it further in the next section together with the reduction of an architecture structure, using AEAR.

Definition 3.5 ***A Normal Architecture Structure has a set of pure parts, a start part, and an end part.***

In this thesis, we suppose that every system architecture structure is normal. If there is a system with more than one start part and more than one end part, it is very simple to add a super-start part and a super-end part, just as the famous Flow Control algorithm does. As in the Figure below, we have a set of start parts $(S_1, \cdots, S_m)$, and a set of end parts $(E_1, \cdots, E_k)$, and we add a super-start part S_0 with availability 1, and a super-end part E_0 with availability 1, and the system becomes a normal structure.

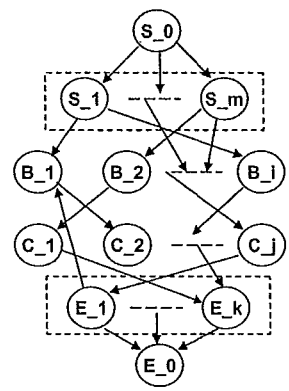

Figure 14: A Variation of Normal Architecture Structure

State Diagram

The state diagram of a normal architecture structure is nearly the same as the normal architecture structure itself, with two additional states **Failure** and **Available** as in Figure 15, which is a state diagram for the architecture structure illustrated in Figure 13. Each state in the diagram corresponds to one part of a normal architecture structure, except for the failure state and available state. We can see that the main purpose of using the state diagram is to evaluate the availability of the software system, and the availability of the software system is related to or dependent on the corresponding architecture parts and the path of their availability checking. So, each state represents the moment that a virtual checker is checking whether this part is available. The transition from one state to another means the checker passed the checking of the availability of the previous part and chose the current part and began to check the current part. For example, an edge from s_i to s_j, means that the checker has passed the availability checking of s_i, and chose part s_j and began to check the availability of s_j. If it failed, or the transition from s_i to s_j failed, this means it has not passed the check of s_i. Please note: *availability checking* here means finding whether the part is available or not, it does not mean the checking of the probability of when the part is available. But the probability of success of passing the current part's availability checking is the availability of that part.

To make sure the entire system with n parts is available, the checker should check all the n parts, this is, there should be at least n transitions from state 1 to state n. And also there is the possibility of repeat checking the same part or transfer to the same state in a state diagram. So, there may be 1 to infinity transitions from state 1 to state n.

Because the next state to be checked will only depend on the availability of current state to pass the check, and it has nothing to do with the history of the availability of the current state, the state diagram has the Markov property. Based on the Markov chain view, as described above, the transition between states is assumed to be a Markov process.

Now, we have a state diagram and we have made clear the relationship between the Markov chain and the state diagram. It is possible for us to define a transition matrix, according to the definition of transition possibility and the state information above. As we defined above, a state diagram is a directed graph, each node S_i represents a state and each directed edge (S_i, S_j) represents a transition from state S_i to S_j. A_i denotes the availability of state S_i of passing the check, that semantically means that the availability of part S_i is A_i. And because the transition (S_i, S_j) only depends on the availability of S_i to pass the check and the probability of choosing S_j to check, and has nothing to do with the history of the availability of S_i or all other states, we say that $A_i V_{ij}$ is the probability for

the transition (S_i, S_j), where V_{ij} is based on the connection E_{ij}, which is defined as,

$$E_{ij} = \begin{cases} 1, & \text{if there is directed edge from } S_i \text{ to } S_j \quad (5a) \\ 0, & \text{otherwise} \quad (5b) \end{cases}$$

So, now our model is, given a set $A = (A_1, \cdots, A_n)$ of all the availabilities of different parts, and an architecture structure, with the Part space $S = (S_1, \cdots, S_n)$, and a set of connections $E = \{(i,j)|$ there is a path from S_i to S_j and $1 \leqslant i, j \leqslant n\}$, then S and E are combined together to construct the state graph (G), which can be described by a matrix:

$$G = \begin{pmatrix} E_{11} & E_{12} & \cdots & E_{1(n-1)} & E_{1n} \\ E_{21} & E_{22} & \cdots & E_{2(n-1)} & E_{2n} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ E_{(n-1)1} & E_{(n-1)2} & \cdots & E_{(n-1)(n-1)} & E_{(n-1)n} \\ E_{n1} & E_{n2} & \cdots & E_{n(n-1)} & E_{nn} \end{pmatrix} \quad (6)$$

where

$$G_{ij} = \begin{cases} 1, & if (S_i, S_j) \in E \\ 0, & otherwise \end{cases} \quad (7)$$

and the out-degree of part i is $d_i = \sum_j G_{ij}$

original transition matrix

Then, without considering part availability, we may obtain the **original transition matrix** V below, according to the out degree d_i of the connecting part.

$$V = \begin{pmatrix} V_{11} & V_{12} & \cdots & V_{1(n-1)} & V_{1n} \\ V_{21} & V_{22} & \cdots & V_{2(n-1)} & V_{2n} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ V_{(n-1)1} & V_{(n-1)2} & \cdots & V_{(n-1)(n-1)} & V_{(n-1)n} \\ V_{n1} & V_{n2} & \cdots & V_{n(n-1)} & V_{nn} \end{pmatrix} \quad (8)$$

where $V_{ij} = \frac{G_{ij}}{d_i}$ means the probability of choosing state S_j from state S_i as the next state to check. It is a random choice, and the probability is the mean proportion of 1, according to the out-degree d_i. Note: the transition probability is independent of the in-degree. This is different from those models for Reliability evaluation in [8,5] because we do not

consider an operational environment; the Markov process here in PBSA is a process for checking whether each state is available; if any part has not passed the check, it goes to a state of Failure. So, choosing the next state for checking is a totally random process.

Then we define the part availability matrix as,

Part Availability Matrix

$$P = \begin{pmatrix} A_1 & 0 & \cdots & 0 & 0 \\ 0 & A_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & A_{(n-1)} & 0 \\ 0 & 0 & \cdots & 0 & A_n \end{pmatrix} \tag{9}$$

And we define the **Basic transition matrix** as

Basic Transition Matrix

$M = P \times V$

$$= \begin{pmatrix} A_1V_{11} & A_1V_{12} & \cdots & A_1V_{1(n-1)} & A_1V_{1n} \\ A_2V_{21} & A_2V_{22} & \cdots & A_2V_{2(n-1)} & A_2V_{2n} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ A_{n-1}V_{(n-1)1} & A_{n-1}V_{(n-1)2} & \cdots & A_{n-1}V_{(n-1)(n-1)} & A_{n-1}V_{(n-1)n} \\ A_nV_{n1} & A_nV_{n2} & \cdots & A_nV_{n(n-1)} & A_nV_{nn} \end{pmatrix} \tag{10}$$

Where M_{ij}=P_r[*successful transition from state* S_i *to* S_j *in one step*] the Basic Transition Matrix must have the following properties:

- $\forall(i,j) \in S \times S, M_{ij} \geqslant 0$
- $\forall(i,j) \in S \times S, M_{ij} \leqslant 1$
- $\forall(i,j) \in S \times S, M_{ij} = A_iV_{ij}$
- $\forall(i,j) \in S \times S, \sum_j M_{ij} \leqslant 1$

where A_i is the availability of part S_i, and V_{ij} is the value of the connection function V.

Adding the states A (Availability) and F (Fault), we get the Markov transition matrix M^{af},

Markov Transition Matrix

$$
T = \begin{array}{c} \\ F \\ A \\ 1 \\ \vdots \\ i \\ \vdots \\ N \end{array}
\begin{array}{c}
\begin{array}{ccccccc} F & A & 1 & \cdots & i & \cdots & N \end{array} \\
\left(\begin{array}{ccccccc}
1 & 0 & 0 & \cdots & 0 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 & \cdots & 0 \\
1-A_1 & 0 & A_1V_{11} & \cdots & A_1V_{1i} & \cdots & A_1V_{1n} \\
\vdots & \vdots & \vdots & \cdots & \vdots & \cdots & \vdots \\
1-A_i & 0 & A_iV_{i1} & \cdots & A_iV_{ii} & \cdots & A_iV_{in} \\
\vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \cdots \\
1-A_n & A_nV_{na} & A_nV_{n1} & \cdots & A_nV_{ni} & \cdots & A_nV_{nn}
\end{array}\right)
\end{array}
$$

This matrix is a Markov transition matrix, and obviously it is a stochastic matrix with property $\bigwedge_i(\sum_k T[i,k] = 1)$, besides the properties of M.

The states F and A are *absorbing* states, because we have $T[FF] = 1 \wedge T[AA] = 1$, and from any state S_i, it is possible to go to F or A (not necessarily in one step). So the Markov chain here is an absorbing Markov chain.

Definition 3.6 *The states F and A are* **absorbing** *states in an absorbing Markov chain for PBSA.*

Definition 3.7 *All states, except F and A, are* **transient** *states in a absorbing Markov chain for PBSA.*

This means all other states are transient, and when $k \to \infty$, we have $M^k[i,j] = 0$, because all the states in M are transient states [1].

Then we can rewrite the Markov transition matrix T as a form as,

$$
T = \begin{array}{c} \\ FA \\ TR \end{array}
\begin{array}{c}
\begin{array}{cc} FA & TR \end{array} \\
\left(\begin{array}{cc}
\mathbf{I} & \mathbf{0} \\
\mathbf{Q} & \mathbf{M}
\end{array}\right)
\end{array}
$$

where $\mathbf{I}$ is a 2×2 identity matrix, $\mathbf{0}$ is a $2 \times n$ zero matrix, $\mathbf{Q}$ is a non-zero $n \times 2$ matrix, and $\mathbf{M}$ is a $n \times n$ matrix. FA represents the states of absorbing and TR represents the transient states.

A standard matrix algebra argument shows that T^n has the form of

$$
T^n = \begin{array}{c} \\ FA \\ TR \end{array}
\begin{array}{c}
\begin{array}{cc} FA & TR \end{array} \\
\left(\begin{array}{cc}
\mathbf{I} & \mathbf{0} \\
* & \mathbf{M^n}
\end{array}\right)
\end{array}
$$

where the $*$ stands for the $n \times 2$ matrix in the left-bottom corner of T^n(This submatrix will be described later.) The form of T^n shows that the entries of M^n give the probabilities for being in each of the transient states after n steps for each possible transient starting state.

According to the theorem 11.3 of [1], we have that in an absorbing chain (with absorbing matrix), the probability that the process will be absorbed is 1. This means that M^n is going to be $\mathbf{0}$ when $n \to \infty$.

Because we do not consider the situation where, after checking the current state, we choose it again as the next state, we have the axiom below:

Axiom 3.4 *The probability of the transition (S_i, S_i) in transition matrix M is 0, so there are no transitions from S_i to S_i, where $S_i \neq A \wedge S_i \neq F$.*

So, $M_{ii} = 0$, where $1 \leqslant i \leqslant n$, in the transition matrix above.

Based on this information, we can determine that the availability checking process in state S_i at time n will be that for state S_j at time $n+k$, using the Chapman-Kolmogorov equations:

$$\forall (i,j) \in S \times S \wedge (n,k) \in \mathbb{N} \times \mathbb{N}, p_{ij}^{(n+k)} = \sum_{m \in S} p_{im}^{(n)} \cdot p_{mj}^{(k)}$$

$$\Longrightarrow M^{n+k} = M^n M^k \tag{11}$$

where $p_{ij}^{(n)}$ means the probability of moving from state S_i to state S_j in n steps.

Now, we say the availability of the whole system obtained from the Markov transition matrix is as follows.

The **availability** *of a software system with a general normal architecture structure is $(I - M)^{-1}[1, n]$, where M is the basic transition matrix of the system, n is the number of the parts, and I is the identity matrix with size $n \times n$.*

proof:

According to the definition 11.3 of [1], given an absorbing Markov chain T, such as the one shown above, we call the matrix $\mathbf{N} = (\mathbf{I} - \mathbf{M})^{-1}$ the *Fundamental Matrix* for $\mathbf{T}$.

And according to the theorem 11.6 of [1], if $\mathbf{B}$ is an $n \times 2$ matrix with entries b_{ij}, where b_{ij} is the probability that an absorbing chain will be absorbed in the absorbing state S_j, from a transient state S_i, then we have

$$\mathrm{B} = \mathbf{NQ} \tag{12}$$

where $\mathbf{N}$ is the fundamental matrix defined above, and $\mathbf{Q}$ is the $n \times 2$ matrix described above in the definition of T.

The proof of this is very easy, that is:

$$\begin{aligned}
\mathbf{B_{ij}} &= \sum_n \sum_k m_{ik}^{(n)} q_{kj} < \quad \text{According to the definition of } B_{ij} \quad > \\
&= \sum_k \sum_n m_{ik}^{(n)} q_{kj} \\
&= \sum_k n_{ik} q_{kj} < \quad \text{Because} \quad \sum_{\mathbf{n}} \mathbf{M^n} = (\mathbf{I}-\mathbf{M})^{-1} = \mathbf{N} > \\
&= (\mathbf{NQ})_{\mathbf{ij}}
\end{aligned}$$

So, it is very clear that b_{1a}, the probability of being absorbed into the absorbing state A from the transient state S_1, will be $(\mathbf{NQ})_{\mathbf{1A}}$=$(\mathbf{NQ})_{\mathbf{12}}$, because state A is in the second column of Q.

So we have the availability of the whole system

$$\mathbf{A} = (\mathbf{NQ})_{\mathbf{12}} = (\mathbf{I}-\mathbf{M})^{-1}[\mathbf{1},\mathbf{n}] \times \mathbf{A_n V_{na}} \tag{13}$$

where I is an identity matrix of size $n \times n$, and A_n is the availability of part S_n and V_{na} is the probability of choosing state A from state S_n.

And in this thesis we assume that there are no edges going out of transient state S_n to other transient states. So we get that $V_{na} = 1$, and $\mathbf{A} = (\mathbf{I}-\mathbf{M})^{-1}[\mathbf{1},\mathbf{n}] \times \mathbf{A_n}$.

This completes the proof.

3.3 Using PBSA

Now, we will show an example of how to calculate the availability of the entire software system, illustrated in Figure 12. The software system of Figure 12 has 12 parts, in which C_1 is the input part and C_{10} is the output part. After doing step one below, we get an normal architecture structure as shown in Figure 13. As we described in last section, we assume that the availability of all parts are now constant, before any updates are done to them, and the transitions will be shown in the state diagram below.

The part availability matrix is:

$$P = \begin{pmatrix} 0.907 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.912 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.956 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.935 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.982 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.903 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.952 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.915 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.983 \end{pmatrix}.$$

Now, we begin to calculate the overall availability of this software system.

Step 1: We reduce the software system architecture to be a normal architecture structure. We will discuss how to do this in next section, using AEAR. The purpose of this is to make sure that the state diagram from this structure will satisfy the qualifications and preconditions of a Markov chain. We assume that we have completed the work and obtained the normal structure as in Figure 13.

Step 2: translate the normal architecture structure from Figure 13 to the state diagram below in Figure 15, because we assume the part-based architecture structure is a normal architecture structure with pure parts, in which all parts have availability independently.

We translate this into a state diagram matrix as

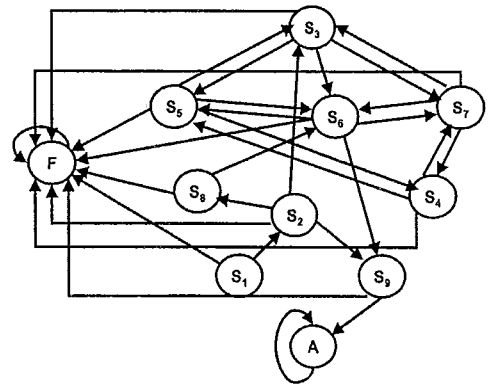

Figure 15: A State View of Software Architecture in Figure 13

$$G = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Then the original transition matrix is:

$$V = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 0 & 0 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 & 1/3 & 1/3 & 1/3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/3 & 1/3 & 0 & 1/3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 1/3 \\ 0 & 0 & 1/3 & 1/3 & 0 & 1/3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Step 3: Model the state diagram by a basic transition matrix according to the state diagram above.

$$M = P \times V$$

$$= \begin{pmatrix} 0 & 0.907 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.304 & 0 & 0 & 0 & 0 & 0.304 & 0.304 \\ 0 & 0 & 0 & 0 & 0.3187 & 0.3187 & 0.3187 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.4675 & 0 & 0.4675 & 0 & 0 \\ 0 & 0 & 0.3273 & 0.3273 & 0 & 0.3273 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.301 & 0 & 0.301 & 0 & 0.301 \\ 0 & 0 & 0.3173 & 0.3173 & 0 & 0.3173 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.915 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

And we get the Markov transition matrix T to be

	F	A	1	2	3	4	5	6	7	8	9
F	1	0	0	0	0	0	0	0	0	0	0
A	0	1	0	0	0	0	0	0	0	0	0
1	0.093	0	0	0.907	0	0	0	0	0	0	0
2	0.088	0	0	0	0.304	0	0	0	0	0.304	0.304
3	0.044	0	0	0	0	0	0.3187	0.3187	0.3187	0	0
4	0.065	0	0	0	0	0	0.4675	0	0.4675	0	0
5	0.018	0	0	0	0.3273	0.3273	0	0.3273	0	0	0
6	0.097	0	0	0	0	0	0.301	0	0.301	0	0.301
7	0.048	0	0	0	0.3173	0.3173	0	0.3173	0	0	0
8	0.085	0	0	0	0	0	0	0.915	0	0	0
9	0.017	0.983	0	0	0	0	0	0	0	0	0

Step 4: Calculate the overall availability of the software architecture. We know that $n = 9$, and

$\mathbf{N} = (\mathbf{I} - \mathbf{M})^{-\mathbf{1}}$

$\mathbf{N[1, n]} = \mathbf{Q[1, 9]} = \mathbf{0.5832}$

So, the overall software availability is:

$A = A_n \times \mathbf{N[1, 9]}$

$= 0.983 \times 0.5832 = 0.5733$

This means the overall availability of this software system is 57.33%.

We will introduce later the AEAR (Availability Equipollence Architecture Reduction), which reduces an architecture structure from any

software architecture with different patterns to a pure part architecture structure, and make it comply with the qualifications and preconditions of PBSA; then, we can use it to calculate the overall availability using PBSA.

3.4 Availability Equipollence Architecture Reduction

3.4.1 Why we need reduction

In a large scale complex system, there may be a lot of patterns embedded in it, the components of some kinds of which may have related availability, which makes the structure conflict with the requirements and preconditions of PBSA and makes the calculation of availability using the PBSA model fail.

Definition 3.8 *A related availability means that a transition from S_i to S_j depends not only on the availability of the previous state S_i, but also on other states S_k, where $k \neq i$.*

In Figure 16, the component-based software architecture includes 4 states, in which state 3 and state 2 are parallel or redundant components. This means if component 2 fails, component 3 will take over its role and make the system available without any breaking (In practice, there should be some other component for monitoring and switching, we omit them to make the example more clear and straightforward).

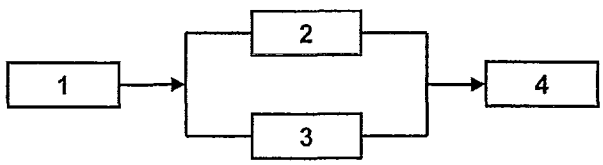

Figure 16: An Example of Related Availability

Suppose that we give the part availability of this software system as

$$P = \begin{pmatrix} 0.907 & 0 & 0 & 0 \\ 0 & 0.912 & 0 & 0 \\ 0 & 0 & 0.956 & 0 \\ 0 & 0 & 0 & 0.935 \end{pmatrix}$$

And the state diagram as:

Translate it into a graph matrix G as:

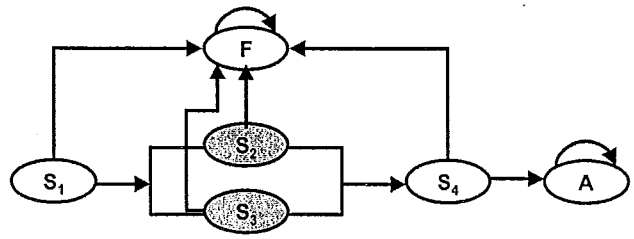

Figure 17: A State View of System in Figure 16

$$G = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

And the original transition matrix is:

$$V = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Then, the Basic transition matrix of this software system is:

$$M = P \times V = \begin{pmatrix} 0 & 0.4535 & 0.4535 & 0 \\ 0 & 0 & 0 & 0.912 \\ 0 & 0 & 0 & 0.956 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

And the markov transition matrix T is:

$$\begin{array}{c} \\ F \\ A \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} F & A & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0.093 & 0 & 0 & 0.4535 & 0.4535 & 0 \\ 0.088 & 0 & 0 & 0 & 0 & 0.912 \\ 0.044 & 0 & 0 & 0 & 0 & 0.956 \\ 0.065 & 0.935 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Then according to PBSA, we have

$\mathbf{N} = (\mathbf{I} - \mathbf{M})^{-1}$

$\mathbf{Q}[\mathbf{1}, \mathbf{n}] = \mathbf{Q}[\mathbf{1}, \mathbf{4}]$

$= 0.8471$

So, the overall software availability is:

$A = A_n \times N[1, 9]$

$= 0.8471 \times 0.9939 = 0.8419$

Obviously, the overall system availability of this software system is now 84.19%.

But according to the simple part availability calculation method introduced in Chapter 3.1, the availability of this system should be

$$\begin{aligned} A_p = & \quad A_1 \times (1 - (1 - A_2)(1 - A_3)) \times A_4 \\ = & \quad 0.907 \times (1 - (1 - 0.912)(1 - 0.956)) \times 0.935 \\ = & \quad 0.907 \times 0.9961 \times 0.935 \\ = & \quad 0.8447 \end{aligned}$$

They are different, so something must be wrong.

Let us look at the transition state diagram. Suppose p_{34} is the transition from S_3 to S_4, and p_{34} obviously depends on the availability of S_3, but it still depends on the availability of S_4, because, in our redundant system, part 2 and part 3 are backups to each other, so if one of them fails, the other will take over its role, timplying there is nearly no break for the overall system. And the unavailability of state 3 will not make the transition p_{34} fail. This conflicts with the property of a Markov chain, in which any transition only depends on the current situation of its previous state, not on its history and not on the other states.

So we have to reduce this architecture structure to an availability equipollent one, in which every part is a pure part, and has no relationship between its availability and that of other pure parts.

Definition 3.9 *An availability equipollent architecture is an architecture structure which has the same availability property as the original software architecture.*

To reduce an architecture structure to an availability equipollent architecture and to a normal architecture structure, in which every part is availably independently, or has no related availability property, we use Availability Equipollence Architecture Reduction (AEAR).

Definition 3.10 *Availability Equipollence Architecture Reduction(AEAR) is a transition process, by which an original architecture structure, with one or more patterns, can be translated to an availability equipollent normal architecture structure.*

Because a software architecture may be heterogeneous, with different kinds of pattern, and a pattern is designed to solve some quality attribute problems, there are some semantically defined relationships among the components in a pattern, and these relationships may lead to related availability properties. So we will introduce approaches for AEAR according to the nature of an architecture pattern. Different patterns may have different rules for AEAR, but there exist some of the patterns, which have same properties for AEAR. So we would like to introduce some approaches for AEAR by classifying the most popular architecture patterns according to their availability properties.

The pattern classes we are going to introduce include: Redundancy/Fault Tolerance/Cluster patterns, Layers/Brokers/Batch-sequential/pipeline patterns, Blackboard/Repository patterns, Parallel/Pipe-Filter patterns, and Interactive/MVC patterns. We will not introduce all patterns, because it is impossible to include all patterns in one paper and actually we doubt if there is a set including all patterns, because every day there are new patterns defined. We class these patterns into 5 classes because each of the classes has a special availability property. The layers pattern, including the client-server pattern and the Browser-Server pattern, are similar to the Brokers, Batch-Sequential and Pipeline patterns. The redundancy pattern is a kind of Fault tolerance pattern, and so is the Cluster pattern.

We are going to introduce 5 kinds of approaches for AEAR, according to 5 classes of patterns, to get availability equipollent normal architecture structures. For each of the patterns, we will present a typical architecture structure for it, and give a superpart for the reduced part of the new availability equipollent normal architecture structure. We will use A_i to represent the availability of original component C_i, and we will use p_{ij} to represent the probability of the transition from state S_i to S_j, where states S_i and S_j correspond to the components C_i and C_j.

[8] introduced two kinds of rules of reduction for simplifying network architecture for reliability calculation. As we mentioned in Chapter 3.1, its rules are based on series and parallel structure. AEAR is another reduction, and its purpose is to make sure that the transition process between two connected components is a Markov process. In addition, AEAR is based on the pattern's structure, and of course it may include the series and parallel structure.

The reduction rules will be named using the pattern name.

3.4.2 Redundancy Rule

Figure 18 is a typical redundancy, fault tolerance or cluster pattern, which consists of a leader component and a set of backup components. There may be different kinds of algorithms or strategies for the monitor/detector, switch decision and recovery, just as we introduced in Chapter 2.3, A simplex structure may be better than this in performance and availability using a different algorithm. But here we will use the most simple and popular one. We suppose that all components are placed in parallel, which means they are executing parallel and, when the leader fails, one of the backup components will be chosen to be a new leader and, without loss of generality, we choose the first backup component as the new leader. In Figure 18, component R_1 to component R_n are backup components to the leader "Leader" component.

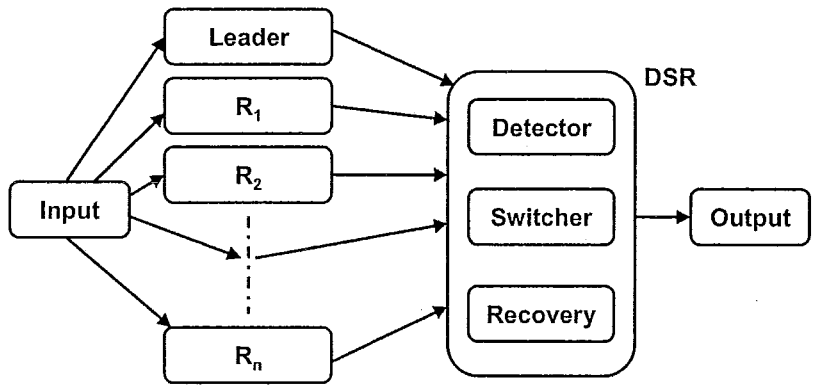

Figure 18: Redundancy or Fault Tolerance or Cluster Pattern

Now, we begin to reduce this structure. To reduce this structure, we should know first that all the components, including the leader component and the backup components have related availabilities, which means that all the components of leader and backups, including the DSR (Detector, Switching and Recovery) component have related availability. So, they should be reduced into one part.

We suppose that the i-th backup component has availability A_i, and the leader component has availability A_l. Then, according to our strategy for this redundancy structure, if the leader component fails, the first backup component R_1 will take over its role, and at this time we should make sure the component R_1 is available, or we should choose the second component R_2 as the new leader component, or $\cdots$. So, the availability A_r of the

reduced part can be:

$$A_r = A_l + (1 - A_l)A_1 + (1 - A_l)(1 - A_1)A_2 + \cdots + \prod_{k=1}^{n-1}(1 - A_l)(1 - A_k)A_n$$

$$= A_l + (1 - A_l)\sum_{k=1}^{n}(\prod_{m=1}^{k-1}(1 - A_m)A_k) \quad (14)$$

The state diagram of the reduced architecture of the original structure, shown in Figure 18, can be like that shown in Figure 19.

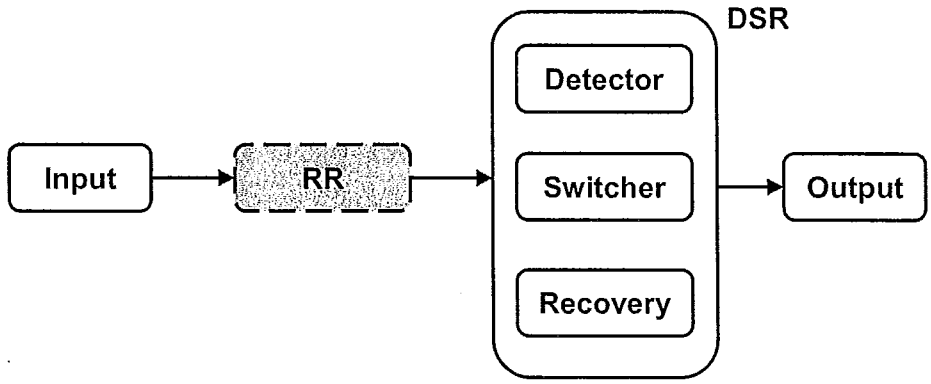

Figure 19: Reduced Structure for Redundancy Pattern in Figure 18

Now, we look back to the example shown in Figure 16. It consists of a redundancy pattern, so we should reduce it into a new state diagram below in Figure 20, from that original state diagram, shown in Figure 17.

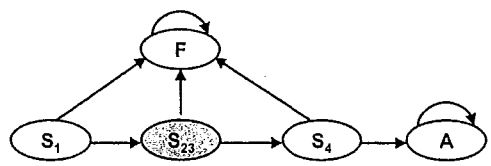

Figure 20: A State View for Reduced Architecture of Figure 16

The reduced superpart here is S_{23}, and its availability can be computed from the equation (14) as:

$$A_{23} = A_2 + (1 - A_2)A_3 = 0.9961$$

So, the part availability matrix should be:

$$P = \begin{pmatrix} 0.907 & 0 & 0 \\ 0 & 0.9961 & 0 \\ 0 & 0 & 0.935 \end{pmatrix}$$

And the state Graph matrix is now:

$$G = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

And the original transition matrix is:

$$V = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

So the basic transition matrix is:

$$M = P \times V = \begin{pmatrix} 0 & 0.907 & 0 \\ 0 & 0 & 0.9961 \\ 0 & 0 & 0 \end{pmatrix}$$

And the Markov transition matrix T is:

$$\begin{array}{c} \\ F \\ A \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} F & A & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0.093 & 0 & 0 & 0.907 & 0 \\ 0.0039 & 0 & 0 & 0 & 0.9961 \\ 0.065 & 0.935 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Then according to PBSA, we have
$\mathbf{N} = (\mathbf{I} - \mathbf{M})^{-\mathbf{1}}$
$\mathbf{N[1, n]} = \mathbf{N[1, 3]} = \mathbf{0.9035}$
And the overall availability of this system is
$A = N[1, n] \times A_3 = 0.9035 \times 0.935$
$= 0.8447 = 84.47\%$
This is the correct availability of this software system.

There are lots of algorithms for redundancy reduction, for example the simplex structure, and the model we introduced in Chapter 2.2.3, first defined by Mario [36]. The cluster model defined by Mario is an excellent one for clusters.

To conduct the availability analysis, they assign each component a failure rate and a repair rate, the rate at which this component recovers from a failure, obtained from questionnaires or checklists, depending on the maturity of the domain or prior experience with similar components. To understand the availability of the RTS, they use a machine repair model with S machines and one repairman. The amount of time each machine operates before breaking down is exponentially distributed with mean $1/\lambda$ (the failure rate of machines is λ). The amount of time that it takes to repair is exponentially distributed with mean $1/\mu$ (the repair rate is μ). Both λ and μ are discovered parameters needed by the availability model.

In the machine repair model [35], they say that the system is in state n whenever n machines are not it use. In a server availability calculation, they say that the system is available (albeit with diminished capacity) whenever at least one server is operating (available) and they say that the system is down whenever all S servers are down. The long-term proportion of time that the system is not in state S (i.e., the system is available, by their definition) is given by

$$A_p = 1 - \frac{S! \cdot (\frac{\lambda}{\mu})^S}{1 + \sum_{n=1}^{S} (\frac{\lambda}{\mu})^n \times \frac{S!}{(S-n)!}} \tag{15}$$

Using this equation, we can reduce a cluster into one availability equipollent part.

Of course, we cannot list all the algorithms here, because there may be new ones every day, but redundancy or clustering is a very good pattern for increasing availability. Further research in this area will be worthwhile.

3.4.3 Sequential Rule

All Layers, Brokers, Batch-Sequential and Pipeline Patterns are executing in a sequential order. Of course, there may be some small difference, for example, Batch-Sequential should obtain its output after all its inputs are fully processed, and pipeline will not have to. Brokers is a kind of variant pattern from Layers, and they are both representative sequential patterns.

The figure below shows the character of a sequential pattern's architecture.

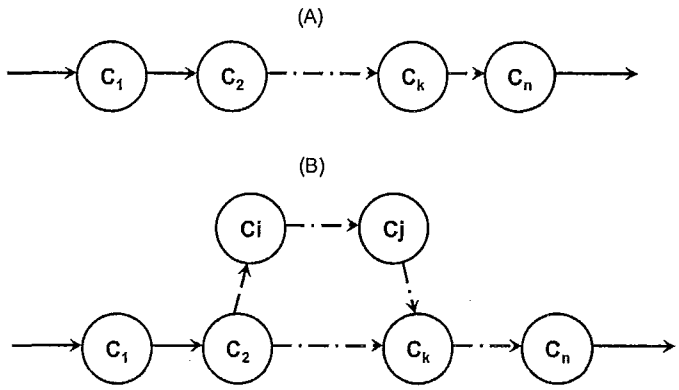

Figure 21: Layers, Brokers and Batch Pattern Architecture

Its very clear that the Figure 21(A) is showing a sequential architecture, just as we introduced in Chapter 2.3.1; the availability of the reduced part can be calculated as

$$A_p = A_1 \times A_2 \times \cdots \times A_n = \prod_{k=1}^{n} A_k \tag{16}$$

The premise of this calculation of the reduced part availability is that, from C_1 to C_n, there are only two connections to outside components, one is the input from outside component to component C_1 and the other is the out-direction connection from C_n to other components not in the set of components which will be reduced.

For example, the following two architecture structures cannot be calculated using equation (16).

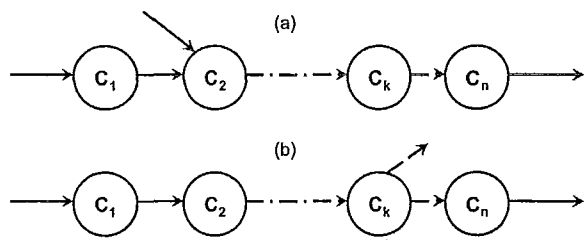

Figure 22: Exceptional Example for Sequential Patterns

In Figure 22(a), the sequential pattern can only start from component C_2, and the reduced availability equipollent architecture should have a structure like:

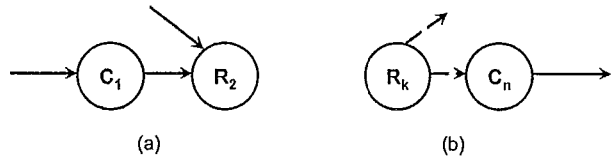

Figure 23: Reduced Structure For Example in Figure22

In the example of Figure 22(a), the reduced structure is shown in Figure 23(a), the part R_2 representing the reduced part for components $C_2, \cdots, C_n$ and its availability is

$$A_{R2} = \prod_{k=2}^{n} A_k \tag{17}$$

In the example of Figure 22(b), the reduced structure is shown in Figure 23(b), the part R_k represents the reduced part for components $C_1, \cdots, C_k$ and the availability is

$$A_{R_k} = \prod_{m=1}^{k} A_m \tag{18}$$

Now, consider the structure shown in Figure 21(b), from component C_2 on, there are branches, each of which is still a sequential pattern; there are no algorithms for redundancy between these branches. This is still a sequential pattern, but a special one.

Let us consider its availability attribute. These branches are choice branches and the execution sequence is decided by component C_2 semantically, so the probability of choosing each branch is decided by the operation profile; precisely, it is decided by the business logic and the class of input data. We will forget about the impact of operation profiles in this thesis and suppose that the probability of choosing each branch is the same.

Now, suppose that in Figure 21(b) there are k branches overall, and each branch i has x_i simple sequential components, each of which has availability of A_{x_i}. Then the availability of the reduced part of this architecture structure is,

$$A_p = \frac{1}{k} \sum_{m=1}^{k} (\prod_{i=1}^{x_m} A_{x_i}) \tag{19}$$

where $\frac{1}{k}$ is the probability of choosing one of the branches.

3.4.4 Parallel Computing Rule

Parallel Computing and Pipe Filter Patterns are totally different patterns from Sequential patterns described above. In modern software system, especially in large concurrent computing systems, including most of the large scale database systems, it is very popular and necessary, because a performance property is the key concern for these software systems. The main difference between these patterns and sequential patterns focuses on the sequence of component execution. In Parallel Computing patterns, a big task comes from a previous component, then it is divided into several logically related subones, each of which is sent to one of the concurrent components to be executed simultaneously, according to a predefined policy. After all the components complete their execution, the combination component will collect all the results from the concurrent components and go to the next component. So, all the concurrent components are executing simultaneously, and they may be working on different processors or working on one processor, by dividing time. The former working style is called the Parallel Computing pattern, the latter is called the Pipe Filter pattern.

But the Sequential Patterns cannot have concurrent components working simultaneously, and all the components should work sequentially.

Now, consider the figure below, showing a Parallel computing pattern.

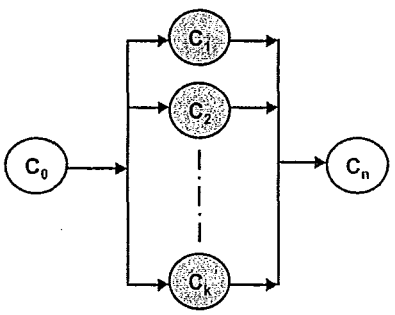

Figure 24: Parallel Computing and Pipe Filter Pattern

There are k concurrent components in this pattern structure, shown in Figure 24. And component C_0 distributes each divided tasks to corresponding components from C_1 to C_k. Then after they finish their computation, component C_n will collect all their results and do some integration and verification work to make sure all components successfully completed their calculation and then integrate all results according to a predefined

policy, then go out .

Now, consider the availability property of this pattern. It is very clear that if there is any fault during the execution of any of the concurrent components, or we say that if one of the concurrent components $\{C_1, \cdots, C_k\}$ fails when executing, then the integration component C_n cannot get all results from component C_1 to C_k, so the component C_n will be always waiting and cannot go further. This means the software system fails.

So, the availability of this pattern is the probability that all the concurrent components are available before the system transfers from component C_0 to them and also they should be available when subtasks assigned to them are being executed. The reduced Parallel Pattern is shown below.

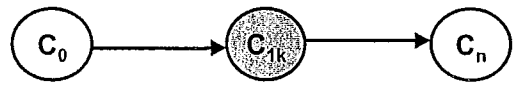

Figure 25: Reduced Parallel Computing Pattern from Figure 24

We suppose all the transitions from component C_0, which will be S_0 if we translate the architecture structure to a state diagram, to the concurrent components will depend only on the availability of component C_0, then the availability of the reduced part(Part C_{1k} in Figure 25) will be

$$A_p = A_1 \times A_2 \times \cdots \times A_k = \prod_{m=1}^{k} A_m \tag{20}$$

We would like to restate that why we can reduce the parallel computing like this is because that we understand that the transition from C_1 to C_n is dependedent not only on the availability of C_1, but also on the component of $C_2, C_3, \cdots, C_k$, in other words, in a state diagram, the transition from S_1 to S_n does not satisfy the precondition of PBSA, so we have to reduce it and make it comply with the precondition, that is to make sure that the transition process from S_i to S_n is a Markov process. And the reduction rule is based on the semantic understanding of the Parallel Computing pattern.

If we have a parallel computing structure, with some concurrent components, from C_1 to C_n, replaced by a sequential pattern, then we can first reduce the sequential pattern into one reduced part, then use the Parallel Computing pattern rule to reduce the overall Parallel Computing pattern.

Note: All the concurrent components should be working in parallel, with no intersection and no branching. Here are two kinds of exceptional

architectures which can not be reduced using Parallel Computing pattern rule.

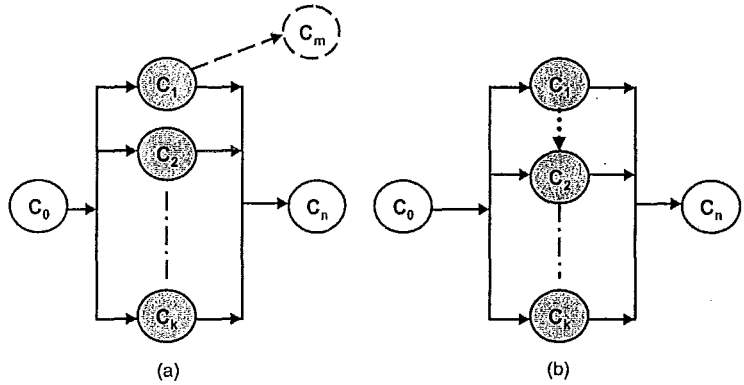

Figure 26: Exceptional Parallel Computing Pattern

The structure of the Parallel Computing pattern is very similar to the structure of the Redundancy Pattern, but they are semantically different. So we must be careful when reducing systems with these two patterns. One may ask how one knows what the pattern is if they have the same structure view. We know what the pattern of the structure is because we are the architects designing the architecture structure and we choose the pattern; so we know which one is the concurrent computing pattern and which one is the redundancy pattern.

This leads to an important conception that an arbitrary, original architecture structure for reduction (AEAR) is a view of components and relationships, which tells us that the relationship between two components or among some components shows not only the connection between two components, but also the semantic relationship between the two or more components, such as the relationships in redundancy and parallel computing.

3.4.5 Blackboard Rule

In the Blackboard, Repository, and Call-and-Return patterns, the execution of a calling component may invoke some services from the other component or components, and after the called component or components

finish executing their job, the controller will come back to the calling component before it goes to other components [4,18].

In Figure 27, there are two examples of Call-Return patterns. The first one, in Figure 27(a), is a typical repository or call and return pattern structure, which has one called component C_2 and offers services for calling component C_1. So, in the state diagram of this pattern, in Figure 28(a), there is a transition from state S_1 to state S_2, and at the same time there is a transition from state S_2 to state S_1.

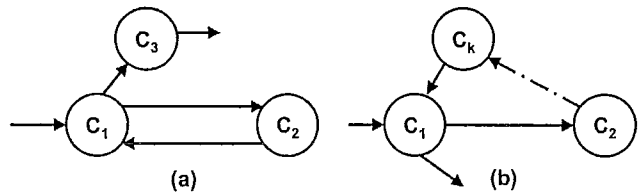

Figure 27: Blackboard and Repository Pattern

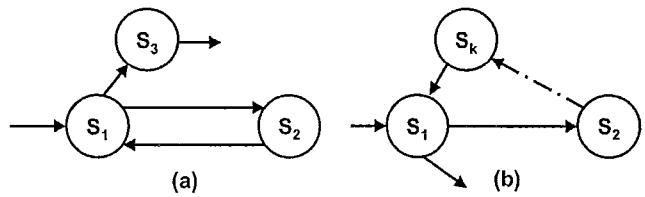

Figure 28: State view of Blackboard Repository Pattern

And Figure 27(b) is a typical Blackboard pattern, where C_1 is a control component, it calls a service of component C_2, and C_2 may call another service on a blackboard component C_3, and then $\cdots C_k$, then C_k returns the result to C_1, then C_1 may recall another service of C_2, and so on. This is a calling cycle.

We suppose that there is no return transition from S_2 to S_1 in Figure 27(a), then it's clear that S_2 and S_1 from a sequential pattern, and its availability is

$A_p = A_1 \times A_2$

Now we add the transition from S_2 to S_1; because the availability of the transition is 1, we say that the availability of the call-return and repository pattern is

$A_p = A_1 \times A_2 \times 1 = A_1 A_2$

Using the same idea, we can obtain the availability of the Blackboard pattern as

$$A_p = A_1 \times A_2 \times \cdots \times A_k = \prod_{m=1}^{k} A_m \tag{21}$$

Then the reduced part can be as in Figure 29(a) and 29(b), respectively.

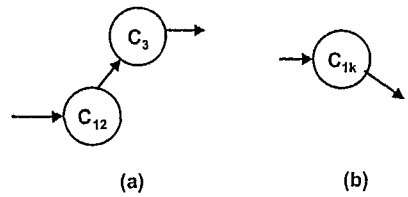

Figure 29: Reduced Blackboard Repository Pattern

where, in Figure 29(a), the reduced part is C_{12}, and the availability of it is A_1A_2 and, in Figure 29(b), the reduced part is C_{1k} and the availability of it is $\prod_{m=1}^{k} A_m$.

Note that, if there is another out-direction connection between C_2 and an other component, other than C_1, as in Figure 30(a), this pattern rule will be broken; this is because it is not a call-return or repository pattern any more. Similarly, if in Figure 30(b), there is an out-direction connection from C_i, where $i \in \{2, \cdots, k\}$, to other components, then the pattern rule will be broken too, and then this will not be a blackboard pattern.

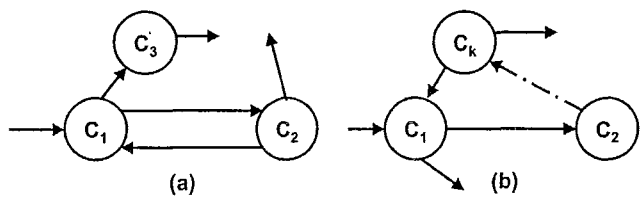

Figure 30: Exceptional Blackboard and Repository Pattern

3.4.6 Interactive Rule

Today's applications requires a high degree of user interaction, many of them achieved with the help of graphical user interfaces. The objective of user interaction is to enhance the usability of an application. Usable software systems provide convenient access to their services, and therefore

allow users to learn the application and produce results quickly [4]. The core property of the architecture of such systems is to keep the functional core independent of the user interface and is based on the functional requirements for the system. User interfaces, however, are often asked to do some changing and adaptation to support different user interface standards, user's feelings and manager's wishes or interfaces that must be adjusted to fit into a customer's business processes. This requires architectures to support the adaptation of user interface parts without causing major effects to application-specific functionality or the data model underlying the software.

There are two particular patterns for interaction systems, the Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC). The MVC pattern provides probably the best-known architectural organization for interactive software systems. It was first introduced in the Smalltalk-80 programming environment [2]. MVC divides an interactive application into the three areas: processing, output, and input. The model component encapsulates core data and functionality and it is independent of specific output representations or input behavior.The view components display information to the user and they obtain the data from the model. Each view has an associated controller component, which receives inputs, usually as events that encode mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system only through controllers.

In Figure 31, there are two scenarios of a MVC pattern, representing two scenarios of MVC. Figure 31(a) shows how user input that results in changes to the model triggers the change-propagation mechanism: changes can be sent to the model to call the services integrated in the model component and then the changes can be propagated to be displayed through the view component. Or the view component requests changes from the model component and displays what the changes by itself. The model component can update the controller component, making its functionality increase or decrease or change.

Figure 31(b) adds references between controller and view components, when initializing. The view component created a controller component for it and the controller component can manipulate the display by directly calling some service on the View component.

The scenario of Figure 31(b) is the structure of initialization of MVC, so we need not use it for calculating availability. Therefore Figure 31(a) is the basic structure for us to calculate availability. We view it as a combined blackboard or repository pattern with a shared component model.

According to the Blackboard rule, we get the availability of MVC

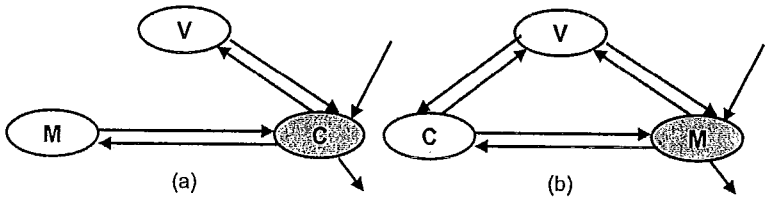

Figure 31: MVC pattern

as,

$$A_p = A_m \times A_c^2 \times A_v \tag{22}$$

If the connection between controller and view components are always there, as Figure 31(b) shows, we have the availability of MVC as,

$$A_p = A_c \times A_m \times A_v \tag{23}$$

Sometimes, we have a MVC structure like Figure 32. Although it is a MVC pattern structure, we cannot use the interactive rule for it.

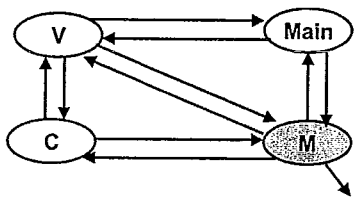

Figure 32: A Special MVC pattern

3.5 Using AEAR

We introduced 5 rules for Reduction and we justified why we need to reduce. So, we now see how we can do an example of architecture availability prediction, using PBSA and AEAR.

Suppose we have the architecture structure shown in Figure 33 below.

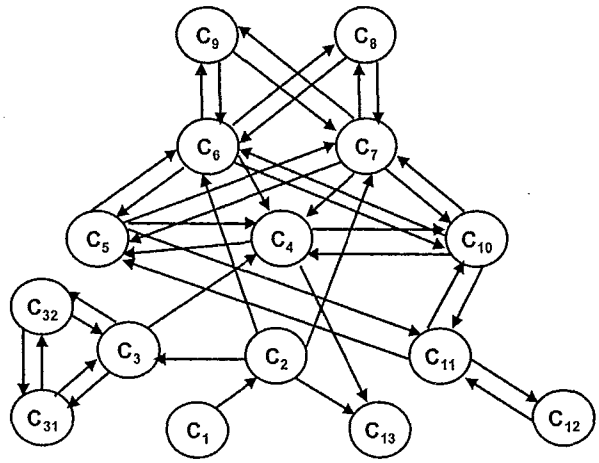

Figure 33: An Example for Using AEAR

The structure shown above is composed of 5 kinds of patterns.

1. Parallel Computing pattern, composed of components C_6 and C_7,
2. Redundant pattern, composed of components C_8 and C_9,
3. Sequential pattern, composed of components C_6 and C_9, or C_7 and C_8,
4. Blackboard pattern, composed of components C_{11} and C_{12},
5. MVC pattern, composed of components C_{31}, C_{32} and C_3 ,

And the availability of these components are as in the table below:

Components	Availability
C_1	0.99987
C_2	0.99999
C_3	0.99987
C_{31}	0.99999
C_{32}	0.99987
C_4	0.99996
C_5	0.99987
C_6	0.99995
C_7	0.99987
C_8	0.99993
C_9	0.99987
C_{10}	0.99929
C_{11}	0.99987
C_{12}	0.99919
C_{13}	0.99929

Now, we have all the resources for calculating the availability of the system.

First, let us reduce the structure using AEAR. As noted above, there are 5 kinds of patterns in the structure, and using the rule we introduce before, we can reduce these patterns into related parts, using corresponding rules.

So, the detailed steps of reduction can be listed as below:

1. Reduce the Redundant pattern, including components C_8 and C_9, into part C_{89}, and the availability of this part is: $A_{89} = 1 - (1 - A_8)(1 - A_9) = 0.999999$.

2. Reduce Sequential patterns, composed of components C_6 and C_{89}, or C_7 and C_{89}, into C_{689} and C_{789}, and using the rule in the sequential pattern, their availabilities are $A_6 \times A_{89}$ =0.999949 and $A_7 \times A_{89}$ =0.999869 respectively.

3. Reduce the Parallel Computing pattern, including components C_{689} and C_{789}, into part C_{6789}, and its availability is: $A_{689} \times A_{789} = 0.999818$

4. Reduce the Blackboard pattern, composed of components C_{11} and C_{12}, into part C_{112}, whose availability is: $A_{11} \times A_{12} = 0.99906$

5. Reduce the MVC pattern, composed of components C_{31}, C_{32} and C_3, into part C_{3x}, whose availability can be calculated as: $A_{31} \times A_{32} \times A_3 = 0.999730$

We have now finished the reduction of the system and we now get the reduced structure in Figure 13, and using PBSA we can easily calculate the availability of this system.

3.6 Extend PBSA to include hardware

We all know that software systems are running on hardware, so the availability of a hardware will obviously influence the availability of software system, as well as the whole system.

Because the failure of a hardware component will automatically lead to a failure of the software component running on it, we can make the hardware a component of the part, which composes the hardware component and the software component running on it in series. Figure 34 shows a software component B, which runs on a hardware component A, so the availability of this part can be calculated using simple part calculation in series:

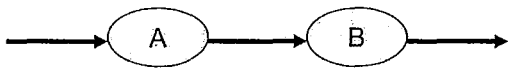

Figure 34: Software component Running on Hardware component

Figure 35(a), part of a redundant pattern, is composed of two software components B and C, running on the same hardware A, we should view this part as a superpart which is composed of a hardware component part and a software part, which is composed of two software components in parallel.

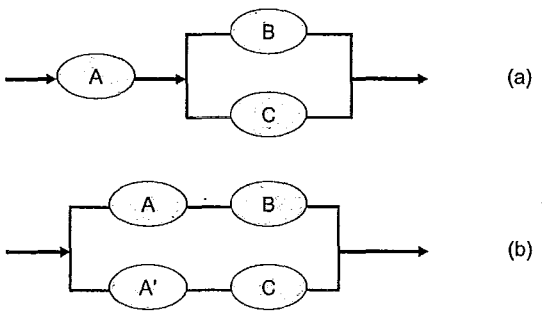

Figure 35: Redundant Pattern in Hardware

so the availability can be calculated as

$$A_p = A_A \times (1 - (1 - A_B)(1 - A_C)) = A_A A_B + A_A A_C - A_A A_B A_C$$

But in Figure 35(b), we have two hardware components, with the same availability A_A, and have two software components running on them,

respectively, and both of the two hardware machines and software components are in a redundant pattern or in parallel. Then the availability can be calculated as

$$A_p = (1 - (1 - A_A A_B)(1 - A_A A_C)) = A_A A_B + A_A A_C - A_A^2 A_B A_C$$

Obviously, the second one (as shown in Figure 35(b)) is better for availability.

In summary, we introduced 5 rules, based on 5 classes of patterns. Each rule has its own character and all the rule are defined according to their semantic definitions.

Some patterns may have the same view of structure, but with different semantic definition. So we have different rules for them. There are still many patterns with their special semantic definition and they may be defined in future works.

4 Strategies for Improving Availability

In this section, we will try to give some strategies for improving the availability attribute of a system. The strategies may include two kinds of methods, Black Box and White Box.

In Black Box method, we will try to do two kinds of work to improve the availability attributes of a system. One is to try to tune the architecture structure of a system, to improve the availability, keep the functionalities, and at the same time get the tradeoff between availability and other quality attributes. The other way is to find the most sensitive component(s) or part(s), and improve the whole system's availability by using redundant component(s) on this(these) sensitive components to improve their availability.

In White Box method, we will try to improve the availability of a component by increasing its reliability and maintainability, and decreasing the detect on time and recovery time for dealing with failures.

4.1 Ranking Component Availability Sensitivity

As we know from the previous section, according to our model, the availability of a system can be calculated using a Markov Transition Matrix. And if we use different variables for availability of different parts, we can represent the system's availability as a formula, composed of all the part availability variables.

Using the example, shown in Figure 15, we let the part availability matrix be like:

$$P = \begin{pmatrix} A_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & A_5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & A_7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_9 \end{pmatrix}$$

Then we get the Markov Transition Matrix:

$$\begin{array}{c} \\ F \\ A \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} F & A & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\ 1-A_1 & 0 & 0 & A_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1-A_2 & 0 & 0 & 0 & A_2/3 & 0 & 0 & 0 & 0 & A_2/3 & A_2/3 \\ 1-A_3 & 0 & 0 & 0 & 0 & 0 & A_3/3 & A_3/3 & A_3/3 & 0 & 0 \\ 1-A_4 & 0 & 0 & 0 & 0 & 0 & A_4/2 & 0 & A_4/2 & 0 & 0 \\ 1-A_5 & 0 & 0 & 0 & A_5/3 & A_5/3 & 0 & A_5/3 & 0 & 0 & 0 \\ 1-A_6 & 0 & 0 & 0 & 0 & 0 & A_6/3 & 0 & A_6/3 & 0 & A_6/3 \\ 1-A_7 & 0 & 0 & 0 & A_7/3 & A_7/3 & 0 & A_7/3 & 0 & 0 & 0 \\ 1-A_8 & 0 & 0 & 0 & 0 & 0 & 0 & A_8 & 0 & 0 & 0 \\ 1-A_9 & A_9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

So, the system's availability, if we consider A_9 to be 1, is:

A=(I-M)$^{-1}$[1, 9]

$= 1/3 \times A_1 \times A_2 \times (2 \times A_6 \times A_8 \times A_5 \times A_3 + A_6 \times A_3 \times A_5 \times A_4 + 6 \times A_5 \times A_3 +$

$2 \times A_6 \times A_8 \times A_7 \times A_3 + A_6 \times A_3 \times A_7 \times A_4 - 6 \times A_6 \times A_3 + 6 \times A_7 \times A_3 +$
$3 \times A_6 \times A_8 \times A_5 \times A_4 + 6 \times A_6 \times A_5 + 9 \times A_5 \times A_4 - 18 \times A_6 \times A_8 +$
$3 \times A_6 \times A_8 \times A_7 \times A_4 + 6 \times A_6 \times A_7 - 54 + 9 \times A_7 \times A_4)$
$/(6 \times A_6 \times A_5 + 6 \times A_6 \times A_7 + 2 \times A_6 \times A_7 \times A_3 + 2 \times A_6 \times A_5 \times A_3 + 6 \times$
$A_5 \times A_3 + 9 \times A_5 \times A_4 - 54 + 9 \times A_7 \times A_4 + 6 \times A_7 \times A_3)$

From A_1 to A_8, if we set all the other variables to be constants, with one variable remaining, for example A_5, we can get a formula for the system's availability like,

$$A = \frac{aA_5 + b}{cA_5 + d} \tag{24}$$

where a,b,c,d are some constants, independent of A_5.

And if we have the variable A_4, we can get the system's availability as,

$$A' = \frac{a'A_4 + b'}{c'A_4 + d'}$$

where a',b',c',d' are some constants, independent of A_4. Now, we want to compare A_5 and A_4, to see which one is more sensitive for the calculation of A. To compare this, we can calculate their derivative to be,

$$d_5 = \frac{ad - bc}{(cA_5 + d)^2}$$

and

$$d_4 = \frac{a'd' - b'c'}{(c'A_4 + d')^2}$$

And then we compare d_5 with d_4, by calculating their difference and using a variable x to represent both variables A_4 and A_5, and then checking which one of d_5 and d_4 is bigger, when $x \in (0, 1]$.

It can be shown, that the availability of a system, calculated using a Markov Transition Matrix, can be represented by the formula similar to (26),

$$A = A_n \times \frac{a_i A_i + b_i}{c_i A_i + d_i} \quad for \quad i = 1, \cdots, n-1 \tag{25}$$

Where n is the number of parts, and a_i,b_i,c_i,d_i are constants, independent of A_i

And the derivatives can be calculated as,

$$d_i = A_n \times \frac{a_i d_i - b_i c_i}{(c_i A_i + d_i)^2} \quad for \quad i = 1, \cdots, n-1 \tag{26}$$

where a_i, b_i, c_i, d_i are constants, independent of A_i.

By, comparing all d_is, we can get a ranked list for all parts of the system; then, we choose the most sensitive one(s), to do some efficient availability improvement work (to be discussed later), and try to obtain the expected system availability.

4.2 Ranking Availability Related Factors

According to the PBSA model, we know that the availability of a system is affected by the following factors,

- The architecture structure of a system, including all kinds of patterns, like redundant patterns;
- The reliability of a component or part;
- The Maintainability of a component;
- The detection time and recovery time of a failure and recovery process.

It may be argued that the last factor should be included in the third one. But we separate them here, because the recovery time here is not the traditional notion of recovery; it may represent the process of taking-over, and the maintainability may include the process of fixing and recovering the failed component or part.

4.2.1 Architecture Structure Tuning

Because the availability of a system is decided by the architecture structure, according to the PBSA model, more precisely speaking, the Markov Transition Matrix being the decision factor for system availability, we can improve system availability by tuning the architecture structure of a system and tuning the structure of the corresponding Markov Transition Matrix.

A very simple example of this kind of tuning is to change two components from running in Series to running in parallel, and the parallel here is the same as redundant. The availability of this part is increased significantly as we mentioned before (may increase from three nines to six nines).

The architecture structure tuning may include:

1. Architecture pattern tuning: tune the architecture pattern and increase the availability of an individual component or part, consequently increasing the whole system's availability.
2. Using the redundant pattern in some sensitive components or parts, to increase the availability of the components or parts, consequently increasing the whole system's availability.

3. Tune the structure of an architecture: tuning the structure of an architecture may lead to the tuning of the Markov Transition Matrix, consequently tuning the availability of the whole system.

The first 2 methods of tuning of the architecture structure may increase availability with no or low risk, and are controllable. But the third one may have some risk, because:

- It may change the system's functionality. For example, if we change two series components into parallel components, and the relationship between these two components are on invoking relation, this will lead to a change to the system's functionality.
- Sometimes, the result of such tuning is unpredictable, because the model of PBSA is based on the structure of an architecture, and based on the Markov Transition Matrix, and it is hard to predict the result of PBSA, when tuning the structure of such a matrix. Sometimes, the result availability after tuning may decrease, and be the opposite of what you wish.
- The structure tuning may affect other quality attributes. Sometimes, one quality attribute is improved, but other quality attributes may be weakened at the same time.

In conclusion, to do the structure tuning, we should

1. Keep the functionality of a system. Even if the availability of a system is your key requirement, and you can get some tradeoff between functionality and availability, it is still hard to decide what the tradeoff is, and it may need to be decided by all stakeholders. And the decision process itself is a complex one.
2. Overall quality attribute evaluation is very important when architecture tuning. Because of the side effect of architecture tuning, the overall evaluation is the promise of getting to tradeoff between availability and other quality attributes. How to evaluate other quality attributes, like Performance and Reliability, is discussed in other works like [5, 24].

Note: We will introduce a new notion of subcomponents, which form a component or part in an architecture structure. We can improve a component's availability by tuning its structure of subcomponents. The method of tuning the structure made up of subcomponents is the same as the tuning of those of components or parts.

4.2.2 Reliability and Maintainability

These two are the other important factors influencing availability, even though they are not as important as the structure of a system. We know that the availability of a component is calculated using the formula:

$$A = \frac{MTBF}{MTTR + MTBF}$$

We see $MTBF$ as some kind of evaluation of Reliability, [8] and [21], and $MTTR$ as some kind of evaluation of Maintainability. Then, we can do an experiment to see which one of Reliability and Maintainability is more sensitive for the increase of Availability. Consider the table below,

Reliability	Maintainability	Availability
500	10	98.04%
500(1+10%)	10	98.21%
500	10(1-10%)	98.23%

If reliability and maintainability are increased by 10%, respectively, the availability is increased in different percentage. Obviously, Maintainability is more sensitive to Availability than Reliability.

Axiom 4.1 *Maintainability is more sensitive to Availability than Reliability, if we see MTBF as Reliability*[5]*, and MTTR as Maintainability*

[Proof:]
Let $MTBF = b$, and $MTTR = c$, then the availability a is

$$a = \frac{b}{b + c}$$

We increase first b by a percentage p and get an availability of

$$a_r = \frac{(1+p)b}{(1+p)b + c}$$

And then increase Maintainability by percentage p, and get another availability of

$$a_m = \frac{b}{b + (1-p)c}$$

[5]There are lots of definitions for Reliability, but we choose MTBF as an easy and comparable way to study them.

Then we compare these two availabilities by calculating their quotient as

$$\frac{a_r}{a_m} = \frac{\frac{(1+p)b}{(1+p)b+c}}{\frac{b}{b+(1-p)c}} = \frac{(1+p)b + (1-p^2)c}{(1+p)b+c} < 1$$

So, we get $a_r < a_m$, which means if we increase the value of Reliability and Maintainability by the same percentage, the availability increases more by the increase of Maintainability than that of Reliability. This means availability is more reactive to Maintainability than Reliability.

4.2.3 Detection and Recovery

Suppose that we have already tuned the structure and used some redundant components for the most sensitive components or subcomponents, then what can we do to improve the availability more?

As we defined above, the availability of a system is the probability that a system is available, or the probability that a system is not in fixing or recovery or failure. So, let us look at a common scenario for a system's failure,

1. A company system fails.
2. The system has redundancy for the most sensitive components.
3. But the system has no detective method for any failures
4. Having received a user's complaint, the administrator of the system realizes that the system failed, and begins to recover using a redundant component. Till now, half an hour has already passed since system failed.
5. The system has no hot backup; the administrator has to rebuild the system's running environment and clear system's log and get the redundant component to the checkpoint at which the system failed. This course of action took 1 hour
6. the system now runs well again, but one and a half hours have passed since the system failed.

Now, let us see what we can do to decrease the unavailable time. The unavailable time focuses on two parts. One of them is detection time, which, in the scenario above, is half an hour. We can find a lot of ways to detect failures of a system, like Heartbeat Failure Detector [6], and Randomized failure detection [7]. And we can also define better components for total failure detection, which means it can not only detect the failure at process

or thread level, it can also detect the level of a component. This will be introduced in the next section.

The other part of unavailable time, in the scenario above, is the recovery time, which is 1 hour in this scenario. There are also a lot of ways to decrease the recovery time, like using hot recovery, realtime checkpoint, etc.

All these methods, used for decreasing detection time and recovery time can make the unavailable time decrease and so increase the availability of a system.

Because the methods mentioned above are related to the structure of a system, we can make this factor part of the structure tuning. So we get a ranking of factors which influence the improving of the availability of a system:

1. Structure Tuning
2. Maintainability improvement
3. Reliability improvement

4.3 White Box Availability Improvement

To improve the availability of an individual component, we can perform three kinds of tasks, as mentioned in the previous section.

1. Structure Tuning
2. Maintainability improvement
3. Reliability improvement

4.3.1 Tune the structure of a component

The most sensitive or effective way is to tune the structure of the component. We may look at a component as a structure, which is composed of subcomponents and all these subcomponents may be connected together as a subsystem. A component is such a subsystem. So we can use PBSA for the evaluation of the subsystem and by using redundancy or tuning the other patterns or the whole structure, we may get better availability.

Note that, redundancy is just one of the structure tuning options, and there may be other options for structure tuning, such as tuning some other patterns, like parallel computing to balance load. This kind of tuning may also improve the availability of a subsystem, a component or part.

But, as mentioned in the previous section, there are some risks when doing such structure tuning. So care must be taken when doing that, because as is often said, *high profit means high risk.*

4.3.2 Get better Maintainability

Software Maintainability evaluation and improvement methods can be accessed from a large amount of literatures, like Welker, who quantified maintainability via a Maintainability Index (MI), in [11], Measurement and use of the MI is a process technology, facilitated by simple tools, that in implementation becomes part of the overall development or maintenance process. These efforts also indicate that MI measurement applied during software development can help reduce life cycle costs. The developer can track and control the MI of code as it is developed, and then supply the measurement as part of code delivery to aid in the transition to maintenance [12].

Maintainability includes two levels of contents, one of which is to enable the system failure to be diagnosed and fixed very quickly, and the other is to enable the system to be updated smoothly.

There is much research and many strategies for improving the maintainability and obtain tradeoffs with other attributes, as in [20].

4.3.3 Get better Reliability

Achieving better reliability is today still a popular topic for many researchers and companies. There is also much research and efforts on it.

[5],[22] and [23] discuss how to evaluate Reliability and [24],[25],[26] discuss how to increase Reliability in different situations.

In summary, the white box availability improvement included some interesting methods and strategies for improving availability. The methods and strategies are based on the PBSA model and based on the ranking of the availability factors. We give strategies here for advising how to choose strategies to improvement our system's availability. The detailed methods and strategies for each of the suggestions are in many of the literatures as listed above or in future works.

4.4 Black Box Availability Improvement

In this section, we will outline some notions about how to improve availability of a system on the component or part level, without digging into the subcomponent level. Of course, one of the ideas about "Redundancy hierarchy" will involve some digging into subcomponent and even process or thread levels.

4.4.1 Hierarchy HA manager components

The HA manager component here is a component, which is embedded into a system at different levels, from the process level, subsystem or component level, to the system level, to monitor the process, components, and system's running situation, do management, diagnosis, failover, data recovery and component switching work. The purpose of this kind of component is to make a system run with high availability.

A typical kind of system structure with an HA manager component can be illustrated as below in Figure 36,

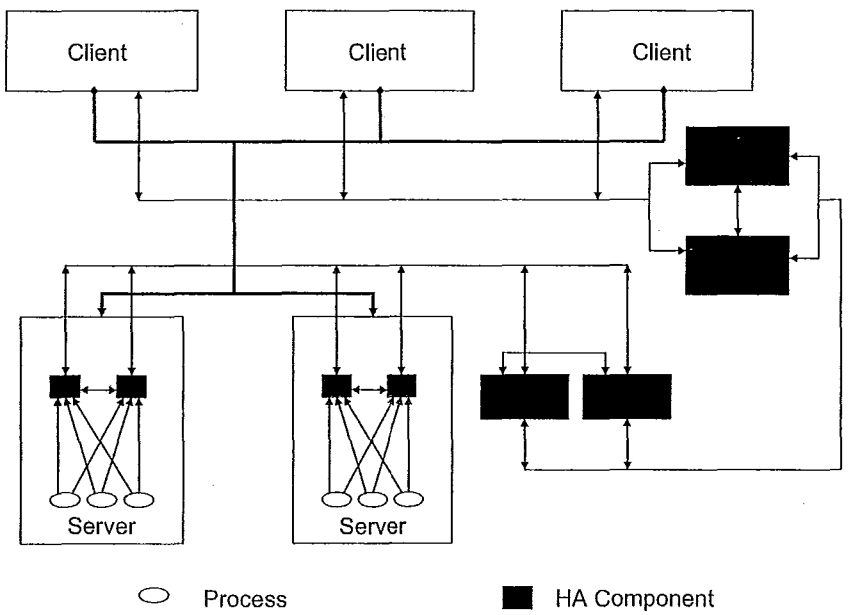

Figure 36: HA Component in a CS System

In this Figure, one may see lots of HA components, This is because we put different levels of HA components in the system, and the size of HA components shown in the figure correspond to the level of the HA components. The lowest level of HA component has the smallest size in Server,

shown in Figure 36. They are used to monitor the running of processes of services in servers of a CS system. If some of the process has a problem, it can quickly detect the failure and switch the role of that process to an other process, according to the existing policy and real-time checkpoint environment. we do not want to describe more details about how to monitor and how to switch the role, and how to set the policy, including the checkpoint interval. The reader can refer to many publications[27],[28], to find such designs or strategies. What we are discussing here is the different levels of HA components and the HA architecture.

Different levels of HA components are organized together to form an HA architecture and different levels have different responsibilities. If there is some problem, which cannot be solved by the lower level HA components, it will be transferred to a higher level HA component. For example, if all the backup process of a server are in deadlock, the HA component, which is directly in charge of them cannot solve the problem, it will ask for help from its higher level HA components, and the higher level HA components may switch the Server itself to another one, according to their existing policy, and if both of the servers are failed, the HA component may transfer the problem to an even higher HA component, and that component may switch the role of local servers to a remote server, according to their existing policy.

This is the notion of Hierarchy HA Components, using which we can save much time for failure detection and recovery, and improve availability.

4.4.2 Redundancy Strategy

Even if we can detect a failure in a very short time, we cannot shorten the recovery time of a system and improve availability if we have no redundancy policy or redundant components for the failed one.

So we use redundancy. As we described in the last section, tuning the architecture structure is the most sensitive factor for improving availability. And redundancy is one of the most effective methods to tune architecture structure. It can improve the availability significantly on number of nines. So, we may say it is the safest way to improve the availability of a system.

Partial Redundancy

The Partial Redundancy Strategy means that we can partially choose some of the most sensitive components to use a redundancy method for the purpose of improving an entire system's availability by improving the availability of these parts or components.

Most systems can realize their availability requirements by partial redundancy. But some of the critical systems, e.g., a satellite launching system, need extreme high availability and reliability, so some of them may

need a full redundancy strategy.

Full Redundancy

A full redundancy strategy means using redundancy for all the components or parts to realize the abnormal requirements of a critical system.

Using a full redundancy strategy requires great economic means. No such an application system using a full redundancy strategy exists.

Redundancy Hierarchy

As noted in the last section, HA manager components can have hierarchical structure, in which all levels of HA manager components are organized together to achieve high availability, and greatly reduced detection time and recovery time.

When HA components at some level detects a failure, they may invoke corresponding recovery procedures or take-over procedures to activate corresponding redundant components or processes. So, sometimes, to achieve extreme high availability, we should set different levels of redundant components, use different levels of redundancy strategies, corresponding to different levels of HA components. This is what we call a hierarchical redundancy strategy.

Just like the full redundancy strategy, the hierarchy redundancy strategy is also a kind of redundancy strategy option for those who need and can afford it. Detailed instructions about how to use it will be left as future work.

4.4.3 Architecture Structure Tuning

Architecture structure tuning is the most sensitive way to improve architecture availability. But, as we mentioned before, there are many limitations and risks exist when we try to do this. If you cannot realize your availability requirements after using all kinds of redundancy strategies, you may need to do this kind of tuning.

So, we classify this strategy as a last option to be chosen, not only because of its difficulty and risk, but also because of its sensitivity.

Detailed research on how to tune the architecture structure according to the PBSA model will be left as future work.

In summary, we have introduced some strategies for black box availability tuning according to the PBSA model. All the tuning should be done not affecting the system's functionality. So we only give some suggestions for availability tuning according to our PBSA model, the detailed work about how to do the tuning work, and how to make sure getting the tradeoffs among all the quality attributes are left as future work.

5 Conclusion and Future work

In this thesis, we reviewed the notion of availability, and the difference between availability and reliability. We gave reasons for why we need high availability and why we need to evaluate availability. We reviewed the relationship between architecture and architectural patterns, and the relationship between architectural patterns and system qualities.

We introduced two qualitative methods, Scenario based method and Markov Model based method, for system availability analysis. These two methods are useful to availability analysis of some components of a system. But they cannot give accurate availability prediction and calculation to a complex system, including all components.

To achieve the availability prediction and calculation function of a complex system, we have defined the PBSA quantitative model, and also defined the AEAR methods and rules for reducing a complex architecture structure to make it comply with the preconditions of PBSA, and then we can use PBSA for availability evaluation.

The PBSA model is based on the Markov model, and the AEAR rules make a system satisfy the preconditions of a Markov Chain. Then the availability prediction model can make us obtain an absorbing Markov Transition Matrix, with two additional states, Failure and Available. The prediction result of the availability of a system comes from the absorbing Markov Transition Matrix naturally.

All AEAR rules come from the semantic definitions of different patterns. These idea comes from much similar research, such as reliability evaluation reduction. But the purpose and method of AEAR are different. We use AEAR for simplifying and reducing the system to make it satisfy the precondition of PBSA, or the precondition of the Markov properties.

Before making any architecture decisions for high availability, we can first evaluate the absolute value of availability by using PBSA, then according to the defined rank of availability sensitivity, and the rank of availability factors, we can choose some strategies to improve availability, including white box and black box strategies.

All strategies for improving availability quality of a system come from the PBSA model. But we haven't given the details about how to plan and implement each strategy. We gave the method, using which we can calculate and compare the sensitivity of the availabilities of all components and obtain the ranking of them. In addition, we gave the ranking of availability factors inside a component, and we gave white box and black box methods for availability improving.

There is still a lot of work left for future research, even the model itself need more research and verification work. such as:

[11] [1995]Welker, Kurt D. Oman, Paul W. "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11 (November/December 1995): 19-23.

[12] [1995]Welker, Kurt D. Oman, Paul W. "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11 (November/December 1995): 19-23.

[13] [2000]Bachmann, F., Bass, L., Chastek, G., Donohoe, P., Peruzzi, F., "The Architecture Based Design Method", Carnegie Mellon University, Software Engineering Institute Technical Report CMU/SEI-2000-TR-001, 2000.

[14] [2000]Bachmann, F., Bass, L., Carriere, J., Clements, P., Garlan, D., Ivers, J., Nord, R., Little, R., "Software Architecture Documentation in Practice: Documenting Architectural Layers", Carnegie Mellon University, Software Engineering Institute Special Report CMU/SEI-2000-SR-004, 2000.

[15] [2000]Bengtsson, P., Lassing, N., Bosch, J., van Vliet, J., "Analyzing Software Architectures for Modifiability", Hogskolan Karlskrona/Ronneby Research Report 2000:11, ISSN: 1103- 1581.

[16] [2001]Rick Kazman. "Software Architecture", Handbook of Software Engineering and Knowledge Engineering,2001

[17] [1999]Mark Klein, Rick Kazman. "Attribute-Based Architecture Styles", CMU. report, October 1999

[18] [2000]Douglas C. Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann, "Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects", Wiley Sons, 2000.

[19] [2001]Len Bass, Mark Klein, and Felix Bachmann. "Quality Attribute Design Primitives and the Attribute Driven design Method", Software Engineering Institute, Carnegie Mellon University.

[20] [2003]Mario Barbacci, Paul Clements, Anthony Lattanze, Linda Northrop, William Wood. "Using the Architecture Tradeoff Analysis MethodSM (ATAMSM) to Evaluate the Software Architecture for a Product Line of Avionics Systems: A Case Study" CMU/SEI-2003-TN-012, July 2003.

[21] [2001] Scott Speaks, "Reliability and MTBF Overview",Vicor Reliability Engineering,2001

[22] [1995] J. Robert Horgan, Aditya P. Marthur, "Perils of Software Reliability Modelings", Feb. 1995

[23] [1999] Wen-Li Wang, Ye Wu, Mei-Hwa Chen. "An Architecture-Based Software Reliability Model", 1999

[24] [1993] N.F. Vaidya, D.K. Pradhan "Fault-Tolerant Design Strategies for High Reliability and Safety ",October 1993 (Vol. 42, No. 10) pp. 1195-1206

[25] [1999] Qiang Xiao, Asce, Robert Sues, Asce, Mark Cesare, Asce and Graham Rhodes, "Computational Strategies For Reliability-Based Multi-Disciplinary Optimization ", 1999

[26] [2002] S. Cliff, NASA Ames, Moffett Field, CA; S. Thomas, Raytheon, Moffett Field, CA; T. Baker, Princeton ,"Aerodynamic Shape Optimization Using Unstructured Grid Methods",University, Princeton, NJ; A. Jameson, Stanford University, Stanford, CA; R. Hicks, , Wickenburg, AZ AIAA-2002-5550 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, Atlanta, Georgia, Sep. 4-6, 2002

[27] [2002] Cheng Zhuo, Yang XiaoHu, Uros Pompe, "High Availability Software Architecture Based on Cluster Technology", Tencon02

[28] [1999]R.K. Iyer, Z. Kalbarczyk, K. Whisnant, S. Bagchi, "A Flexible Software Architecture for High Availability Computing",

[29] [1997]G.Abowd, L.Bass, P.Clements, R.Kazman, L.Northrop, "Recommended best industrial practice for software architecture evaluation". Technical Report CMU/SEI-96-TR-025 and ESC-TR-96-025, January 1997.

[30] [2001]Len Bass, Mark Klein, Gabriel Moreno, "Applicability of General Scenarios to the Architecture Tradeoff Analysis Method", CMU/SEI-2001-TR-014, October 2001

[31] [1996]Shaw, M., Garlan, D., "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall, 1996.

[32] [1994]Kazman, R., Bass, L., Abowd, G., Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of ICSE 16, Sorrento, Italy, May 1994, 81-90.

[33] [1994]D. DAWSON, "INTRODUCTION TO MARKOV CHAINS", McGill University

[34] [2006]Richard G. Clegg, "Modelling internet trac using Markov chains", Imperial College, May 2006

[35] [1989]Ross, S. M. "Introduction to Probability Models", 4th Edition. Boston, MA: Academic Press, 1989.

[36] [1998]Mario R. Barbacci, S. Jeromy Carriere, Peter H. Feiler, Rick Kazman, Mark H. Klein, Howard F. Lipson, Thomas A. Longstaff, Charles B. Weinstock "Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis ", May 1998

[37] [2006]XoSoft "Business and IT Requirements for Continues Data Protection", April 2006