CHAPTER 1
INTRODUCTION

With the advent of the computer age, computers, as well as the software running on them, are playing a vital role in our daily lives. The software industry is booming exponentially. Software and its usage has become ubiquitous from the humble digital watches to rocket science, its incomprehensive today to think of technology advances without software of some kind. We seldom see any industry or service organization working without the help of an embedded software system. Software is embedded in all kind of systems such as transportation, medical, telecommunications, military, industrial processes, entertainment and the list is almost endless. The continuously growing demand of large and complex software systems has introduced major challenges for the software developers’ viz. quality, cost effectiveness, reliability and delivery time of the software. Demand of large and complex systems on the other hand gave birth to Component Based Software Development (CBSD).

A component based software system is usually built up of multiple software modules such that each module is more manageable. Module design is identified during the development process and integrated later to form a complete system. The modular design is even further sub divided into smaller components such that these small components can be developed independently and later linked to develop a full functionality. The developers have different options for the development of these small independent components such as choosing from already available commercial off-the-shelf (COTS) components, in-house development, or modifying the functioning of some existing components. By using COTS components the productivity of development process is improved in terms of time, costs and reliability.

Reliability of computer systems has become a major concern for software industry. Techniques of fault avoidance and fault removal during the development phase allows the developers to build highly reliable software, however even then one can’t assure that the software would never fail in operational phase. There comes the need of software fault-tolerance techniques which enable a software system to tolerate software faults remaining in the system after it is put to use.
The decision to choose right mix of components for a CBSD is not an easy task, it involves a trade-off between quality and cost along with many other concerns. Manager cannot take this decision based on his intuition, or the past experiences. He has to rely upon the scientific way of decision making. Mathematical modeling and optimization is the scientific field of Operational Research offers promising solution to this decision problem. This thesis endeavors to apply techniques of Operational Research to modeling and optimization in component selection for a Component based software engineering for fault tolerant modular software systems under consensus recovery block scheme.

This chapter is introductory chapter and divided into two sections. It gives a brief description of various concepts used in the thesis and literature survey. The topics included in the First Section are software engineering concepts, software development, component based software engineering and fault tolerant software systems. Topics on operational research, optimization and fuzzy optimization are also included in this section. Second Section discusses the literature review on optimal component selection for a modular software system.

1.1 EVOLUTION OF SOFTWARE ENGINEERING AS AN ENGINEERING DISCIPLINE

In the era of globalization every sector of economy depends on computer to a great extent. Software is a key element of any computer based system. Industries have realized that quality software is an important means to improve performance and to have competitive advantage. The development and supply of software is one of the fastest growing industry segments in India and abroad. Today, computers are widely used in air-traffic control, nuclear reactors, government administration, and business management and also in a wide range of service sectors such as banking, transport, education, entertainment, healthcare etc. This has been made possible due to the rapid developments in the field of Information Technology (IT). IT has opened a new era of information for today’s organization and has enabled it to gather and communicate information and thus function in a more effective way. IT products and services have brought a revolutionary change in the world. It has benefitted society and increased the
global productivity, but a major threat of this revolution is that the world has become critically dependent on the computing systems for their proper functioning and timing of all activities. The software industry is continuously facing a challenge of developing effective large-scale software systems. Various concepts and methods such as abstract data types, structured programming, design patterns, modeling languages etc., have been employed to improve the effectiveness of developing software systems.

Functionality of computer operations has become more essential and more complicated. Critical applications have increased in size and complexity. There is a great need for looking at ways to quantify and predict the reliability of computer systems in various complex operating systems [Pham, (2006)]. The cost and consequences of these systems failing can range from inconvenience (e.g., malfunctions of home appliances) to economic damage (e.g., interruptions of banking systems) to loss of life (e.g., failures of flight systems or medical software). This has increased our dependence on machines and its reliability. The idea of unreliable software is unimaginable and damaging. Projects are sometimes year late. They cost much more than originally predicted, are unreliable, difficult to maintain and performed poorly. Software failures have led to serious consequences in business. Many companies have experienced failures in their accounting system due to faults in the software itself. The failures range from producing the wrong information to the crashing of whole system. Another well-known example is of Ariane-5, where the European Space Agency took 10 years and $7 billion to produce the so called giant rocket capable of hurling a pair of three ton satellites into orbit with each launch. It was intended to give Europe overwhelming supremacy in the commercial space business. The rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles along with its payload of four expensive and uninsured scientific satellites [Aggarwaal, (2007)]. Many of the problems with software development could be traced to faulty methodologies. These problems of software development were extensively felt in the beginning of the 1970’s and are collectively referred to as software crisis. This prompted managers, software professionals and researchers to analyze the cause of these problems. They felt the need to adopt a systematic approach to software development. Application of, principles of engineering for systematic development of software gave rise to a new discipline called “Software Engineering”. The immediate concern of software engineering
was aimed at scheduling and systematizing the software development process to have a control over the various stages of software development using its tools, methods and process to engineer quality software. To gain understanding of software engineering it is important to know about the evolving role of software and its characteristics.

1.1.1 The Evolving Role of Software

Software has witnessed many changes since its inception. It has evolved over time in spite of all odds and adverse circumstances. Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. As a vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), the creation and control of other programs (software tools and environments).

Software engineers are concerned with developing software products, i.e. software which can be sold to a customer. There are two types of software product:

1. **Generic Products:** These are stand-alone systems which are produced by a development organization and sold on the open market to any customer who is able to buy them. Sometimes they are referred to as shrink-wrapped software. Examples of this type of product include databases, word processors, drawing packages and project management tools.

2. **Customized Products:** These are systems which are commissioned for a particular customer. The software is developed specially for that customer by a software contractor. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

An important difference between these different types of software is that, in generic products, the organization which develops the software controls the software
specification. For customized products, the specification is usually developed and controlled by the organization that is buying the software. The software developers must work to that specification.

1.1.2 Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form [Pressman, (2005)]. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware.

1. **Software is developed or engineered; it is not manufactured in the classical sense.**

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent for software. Both activities depend on people, but the relationship between people applied and work accomplished is entirely different. Both require the construction of a “product.” But the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. **Software doesn’t wear out.**

There is a well-known “bath-tub curve” in reliability studies for hardware products. Figure 1.1.1 depicts the failure rate as a function of time for hardware. Hardware products exhibits very high failure rates early in the life cycle. Defects are then corrected and failure rates will drop to a steady state and after reaching its wear-out period where the hardware components start failing.
Software is not susceptible to the environmental maladies that cause hardware to wear-out. Therefore, in theory, the failure rate curve for software should take the form of the “idealized curve” shown in figure 1.1.2. Undiscovered defects will cause high failure rates early in the life of a program, however, these are corrected and curve flattens as shown. The failure will increase only when new release is introduced. However, the implication is very clear, software does not wear out. For every software change/release, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in figure 1.1.2. Before the curve can return to the original steady state failure rate, another change is requested causing to spike again.

3. Component based software development.

As an engineering discipline evolves, a collection of standard design components is created. The reusable components have been created so that engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In a software world, it is gaining importance day by day. In software every project is a new project. We start from the scratch and design every component of a software system. Huge effort is required to develop software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. These standard components are easily available with the commercial vendors and can be bought and integrated to form a complete system. This leads to a development of a component based software system.
4. Hardware versus Software changes

Incorporation of changes or modifications in the requirement is an important and critical activity in building hardware as well as software systems. Sometimes, it becomes obsolete and needs to be replaced or requires radical modification, as functional and performance requirements change. The changes made in one component of the hardware system do not affect the other components of that system. But when a change is made in one component of a software system then this change may be reflected in the other components as well. Therefore, for all changes desired by the customer or user, the software engineer has to respond effectively by taking into account the design and architecture of the software.

5. Hardware versus Software Failure

When the hardware component fails in its operation, then that can be corrected by either repairing or replacing that component. The hardware system so obtained remains the same as it provides the same functionality after it gets repaired. But this is not true for the software systems. A fault in the software system leads to its failure. When a software system fails to perform the desired tasks then this is treated as a failure by the software developer as well as by the user. When this happens the software engineer detects the cause of the software failure, incorporate the change or modify the existing code and executes the software according to the desired specifications. Therefore, software system so obtained is a new software.

1.1.3 Software Engineering

The computer revolution has benefited society and increased the global productivity, but a major threat of this revolution is that the world has become critically dependent on the computing systems for proper functioning and timing of all its activities. Software Engineering emerge as a discipline to develop methods and procedures for software development that can scale up for large systems and that can be used to produce high quality software at low cost and short delivery time.
An engineering discipline is driven by practical parameters of cost, delivery time and quality. Like all engineering disciplines, software engineering is also driven by three major factors: cost, delivery time and quality. The cost of developing a system includes manpower, purchase price for software components, cost involved in building a software component, and other support related cost. Delivery time is an important factor in many projects. The manager of the software development team has to schedule all the activities in such a way so that the final software product is delivered in a short period of time. One of the major factors driving any production discipline is quality. In the current times, quality is the main focus, and most business strategies are designed around it. Developing methods that can produce high-quality software is another fundamental goal of software engineering.

1.1.4 Software Reliability Engineering

Software Reliability Engineering (SRE) is centered on a very important software attribute—reliability. Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [ANSI/IEEE, (1991)]. Software Reliability Engineering broadly focuses on quantitatively characterizing the following standardized six quality characteristics defined by ISO/IEC: functionality, usability, reliability, efficiency, maintainability and portability. Software reliability is the major dynamic attribute of software quality, quantifies software failures—the most unwanted
events, and hence is of major concern to the software developers as well as users. As a proven technique, SRE has been adopted either as standard or as best current practice by the organizations in their software development projects including AT&T, Lucent, IBM, NASA, Microsoft and many others.

Applications of SRE principles on any software development project provide several simultaneous benefits. It not only provides enough reliability but also avoid both excessive cost and development time. SRE helps in the following:

- Analyse, manage, and improve reliability of the software
- Balance customer needs for competitive price, timely delivery, and a reliable software product/project
- Determine when the software is good enough to release to the customer, minimizing the risks of releasing software
- Avoid excessive time to market due to over testing

Conceptually SRE is a layered technology (figure 1.1.3). It rests on the organizational commitment to quality with a continuous process improvement culture and has its foundation in the process layer. Process defines the framework for management control of the software projects, establishes the context in which technical methods are applied, work products are produced, quality is ensured and change is properly managed. SRE method provide the technical “how to’s” for building the software whereas the tools provide automated or semi-automated support for the process and methods [Pressman, (2005)].

[Source: Pressman, (2005)]

Figure 1.1.3: Software Reliability Engineering Layers
Figure 1.1.3 shows the relationship between principles, methods, methodologies, and tools. Each layer in the figure is based on the layer(s) below it and is more susceptible to change, due to the passage of time. This figure shows clearly that principles are the basis of all methods, techniques, methodologies, and tools.

Design of proper software development processes and their control becomes primary goal of software reliability engineering. A development process consist of various phases, each phase ending with the defined output. The phases are performed in an order specified by the process model being followed. Most of the SDLC models has to undergo the following phases: feasibility study, requirements and specification, software design, coding, testing, system delivery and maintenance. Tools and techniques of software reliability engineering in a phased process allow proper checking for quality and progress throughout the SDLC.

### 1.1.5 Software Development Life Cycle

The **Software Development Life Cycle (SDLC)** is a process used to facilitate the development of a large software product in a systematic, well-defined, and cost-effective way. An information system goes through a series of phases from conception to implementation. Various reasons for using life-cycle models include:

- Helps to understand the entire process
- Enforces a structured approach to development
- Enables planning of resources in advance
- Enables subsequent control of them
- Aids management to track progress of the system

While developing a software product, it is necessary for the development team to identify a suitable life cycle model and adhere to it. The advantage of adhering to a life cycle model is that it encourages development of software in a systematic and disciplined way. It breaks the problem of developing software into successfully performing a set of phases, each handling a different concern of software development.
This ensures that the cost of development is lower than what it would have been if the whole system was tackled together. Furthermore, a phased process allows proper checking for quality and progress at some defined points during the development (end of phases). A phased development process is central to the software engineering approach for solving the software crisis. Therefore, life cycle models help to achieve the following objectives:

- **Improvement and guarantee on the quality**
- **The cost for the whole life cycle can be checked**
- **Communications between the different parties is improved, and there is a reduced in the dependence of customer on the contractor**

Each process model has strengths and merits that make it suitable for specific application. However all processes are the same, so far as some activities are concerned. These activities by default are required to be carried out whether the software is simple or complex, small or large and may belong to any type. The software models comprise activities, tasks and deliverables. The generic process framework applicable to vast majority of software projects include the following activities:

- **Planning, scheduling, tracking and control of activities**
- **Technical reviews, designs, architecture and programs**
- **Software quality assurance**
- **Documentation**
- **Risk analysis**
- **Measurements of efforts, resources, costs and budgets for planning and building development standards**

Almost all known process models bear at least some similarity to the preliminary process models known as waterfall model which was proposed by Royce in (1970). The framework activities are shown in figure 1.1.4 and can be illustrated as follows
Activity 1: Feasibility Study

The aim of feasibility study is to determine whether developing the product is financially and technically feasible. An estimate is made of whether the identified user needs may be satisfied using current software and hardware technologies. The study will decide if the proposed system will be cost effective from a business point of view and if it can be developed given existing budgetary constraint. A feasibility study should be relatively cheap and quick. The result should inform the decision of whether to go ahead with a more detailed analysis.

Activity 2: Requirement Analysis and Specification

The aim of the requirements analysis and specification phase is to understand the exact requirement of the customer and to document them properly. This is accomplished with frequent interaction with the users. The goal of the requirement analysis is to collect and analyze all related data into information with a view to understand the customer requirements clearly and weeding out inconsistencies and incompleteness in these requirements. During this phase, the user requirements are properly organized and documented in a SRS document which is the final outcome of this phase. It includes collection of product functionality, usability, intended use, future expectations, user environment and operating constraints. The emphasis in requirement phase is on
identifying what is needed from the system, and not how the system will achieve its
goal. The engineer doing requirement analysis and specification is usually designated
as analyst.

**Activity 3: System Design**

The goal of the design phase is to transform the requirements specification into a
structure that is suitable for implementation in some programming language. This phase
is the first step in moving from the problem domain to solution domain. In other words,
starting with what is needed design takes us towards how to satisfy the needs. The
design of a system is perhaps the most critical factor affecting the quality of the
software. The output of this phase is the design document. The design activity is often
divided into two separate phases—software design and detailed design. System design,
which is sometimes called top-level design aims to identify the modules that should be
included in the system, how they interact with each other to produce the desired results.
During detailed design the internal logic of each of the modules specified in system
design is decided. Therefore, the design engineers must take care that the designed
system architecture, program structure and algorithm design conforms to the
specification document. Any glitch in the design phase could be very expensive to
solve in the later stage of software development.

**Activity 4: Coding**

The program structures and algorithms specified in the design document are coded in
some programming language - a hardware readable form. This phase consist in
identifying existing reusable modules, coding of new modules, modifications in
existing modules, code editing, code inspection and a final test plan preparation. If the
program design is performed in a detailed manner, code implementation can be
accomplished without much complication. Programming tools like compilers,
interpreters and debuggers are used to generate the code. Different high level
programming languages like C, C++, Visual basic and Java are used for coding. With
respect to the type of application, the right programming language is chosen. Once the
independent programs are implemented they are linked to form, the modular structure
of the software according to the interface relations defined in the design document.
Activity 5: Testing

Testing is the verification and validation activity for the software product. The goals of the testing phase are:

- To affirm the quality of the product by finding and eliminating faults in the program.
- To demonstrate the presence of all specified functionality in the product.
- To estimate the operational reliability of the software.

During the testing phase, program components are combined into the overall software code and testing is performed according to developed test (software verification and validation) plan. The testing phase consist of:

1. Unit test
2. Integration test
3. Acceptance test

![Software Testing Diagram]

**Figure 1.1.5: Software Testing**

**Unit test** is the process of taking a program module and running it in isolation from the rest of the software product. By using prepared inputs and comparing the actual results with the results predicted by the specifications and the design of the module.

**Integration test** includes sub-system and system tests. The sub-system test focuses on testing the interfaces and independencies of sub-systems or modules. The system test
tests all the modules as a whole to determine whether specified functionality is performed correctly as the results of the software. The integration test also includes the system integration process which brings together all system component hardware, software and humanware. This testing is conducted to ensure the system requirements in real or simulated system environments, are satisfied.

**Acceptance test** acts as a validation of testing phase, consisting of internal test and field test. The internal test includes capability test and guest test, both performed in-house. The capability test tests the system in an environment configured similar to the customer environment. The guest test is conducted by the user in their software, organization sites. The field test is to install the product in the user environment and allows the user to test the product where customers often lead the test and define and develop the test cases. The field test is also called “beta test”. The acceptance test is defined as formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable the customer to determine whether the system is acceptable.

**Activity 5: Operation and Maintenance**

The system or system modifications are installed and made operational in the operational environment. The phase is initiated after the system has been tested and accepted by the user. Installation also involves user training primarily based on major system functions and support. The users are also provided installation and operation manuals. This phase continues until the system is operating in accordance with the defined user requirements. Inevitably the system will need maintenance during its operational use. During this period the software is maintained by the developer to conquer the faults that remain in it at its release time. Software will definitely undergo change once it is delivered to the customer. There are many reasons for a potential change. Change could happen because of some unexpected input values into the system. Changes in the system could directly affect the software operation. The software should be developed to accommodate changes that could happen during the post implementation period.

**1.1.6 Software Reliability**

Software developers aim at building software with minimum resources and simultaneously achieving a maximum reliability. The resources are not available in
abundance, therefore, restrictions have to be placed on manpower, cost, delivery time, etc. so as to build a product in accordance with the user requirements.

**Software Reliability – IEEE Definition**

Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time.

Before we further proceed with the discussion on software reliability, we first define two commonly used terminologies in the area of software development, viz., Fault and Failure. **Failure** is defined as the deviation of the actual output from the expected output. **Fault** is a defect in the program that, when executed, causes a failure. There are many real life examples when faults in computer systems of safety critical systems have caused spectacular failure resulting in calamitous loss of life and economy. Therefore the reliability of computer systems has become a major concern for software industry.

Means to cope with the existence and manifestation of faults in software are divided into three main categories:

- **Fault avoidance/prevention:** This includes use of software design methodologies, which attempt to make software provably fault-free.
- **Fault removal:** These methods aim to remove faults after the development stage is completed. Exhaustive and rigorous testing of the final product does this.
- **Fault tolerance:** This method makes the assumption that the system has unavoidable and undetectable faults and aims to make provisions for the system to operate correctly even in the presence of faults.

Fault avoidance during the design phase and fault removal during the testing phase facilitate in improving the reliability of the system to a desired level. Still the software is suspected to contain faults due to the inability of exhaustive system testing. These faults result in software failure during the field usage if executed. Now a day’s software is used to automate and control the functioning of many critical applications as defense, spacecrafts, air traffic control, nuclear power plants, banking systems and many more. It is very important to provide very high reliability in the software developed for these applications as a failure can result in catastrophic results. For safety critical systems it is necessary to
build an inbuilt ability of survival into the system even on a software failure. The effective way to achieve this property is to design a fault tolerant system. For example the Computer Aided Traffic Control System is fault tolerant software designed to control the Japanese Railways [Pham, (2006)]. A fault tolerant system assures improved reliability by using protective redundancy at system level. A careful use of redundancy may allow the system to tolerate faults generated during software design and coding thus improving software reliability.

1.1.7 Fault-Tolerance

Fault-tolerant systems are defined as systems capable of recovery from software failure to provide uninterrupted real-time service. Fault-tolerant software assures system reliability by using protective redundancy at the software level. There are mainly two strategies for software fault tolerance - error processing and fault treatment. Error processing aims to remove errors from the software state and can be implemented by substituting an error-free state in place of the erroneous state, called error recovery, or by compensating for the error by providing redundancy, called error compensation. Error recovery can be achieved by either forward or backward error recovery. The second strategy is, fault treatment, it aims to prevent activation of faults and so action is taken before the error creeps in. The two steps in this strategy are fault diagnosis and fault passivation. Figure 1.1.6 shows this classification of fault tolerance systems. The nature of faults, which typically occur in software, has to be thoroughly understood in order to apply these strategies effectively.

Techniques for tolerating faults in software have been divided broadly into three classes – (1) design diversity, (2) data diversity and (3) environment diversity. Table 1.1.1 shows the fault tolerance strategies used by these classes.

1.1.7.1 Design Diversity

Design diversity techniques are specifically developed to tolerate design faults in software arising out of wrong specifications and incorrect coding. The method requires redundant software elements that provide alternative means to fulfill the same specifications. The aim is to obtain system survival on some input by the means of a correct output from at least one of the alternatives hence, no system failure on most occasions. Software reliability engineering technique suggests using different
specifications; design, programming languages and team, algorithms to build the alternative versions so that the independent version fails in independently with least possible common failures. These variants are used in a time or space redundant manner to achieve fault tolerance. Popular techniques, which are based on the design diversity concept for fault tolerance in software, are-Recovery Block, N-Version Programming and N- self Checking Programming. Few Hybrid schemes are also proposed by some researchers [Avizienis and Kelly, (1984)].

![Fault Tolerant Strategies for Software](image)

Table 1.1.1: Strategies used by Different Fault Tolerance Methods

<table>
<thead>
<tr>
<th>Technique</th>
<th>Design Diversity</th>
<th>Data Diversity</th>
<th>Environment Diversity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error compensation</td>
<td>Yes</td>
<td>Yes</td>
<td>–</td>
</tr>
<tr>
<td>Error recovery</td>
<td>Yes</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Fault treatment</td>
<td>–</td>
<td>–</td>
<td>Yes</td>
</tr>
</tbody>
</table>

[Source: Kapur et al, (2011)]

1.1.7.2 Data Diversity

The technique of data diversity, a technique for fault tolerance in software, was introduced by Amman and Knight, (1988). The approach uses only one version of the software and relies on the observation that a software sometime fails for certain values in the input space and this failure could be averted if there is a minor perturbation of input data which is acceptable to the software. N-copy programming, based on data diversity, has N copies of a program executing in parallel, but each copy running on a
different input set produced by a diverse-data system. The diverse-data system produces a related set of points in the data space. Selection of the system output is done using an enhanced voting scheme, which may not be a majority voting mechanism. This technique might not be acceptable to all programs since equivalent input data transformations might not be acceptable by the specification. However, in some cases like a real time control program, a minor perturbation in sensor values may be able to prevent a failure since sensor values are usually noisy and inaccurate. This technique is cheaper to implement than design diversity techniques.

1.1.7.3 Environment Diversity

Environment diversity is the newest approach to fault tolerance in software. Although this technique has been used for long in an ad-hoc manner, only recently it has gained recognition and importance. Having its basis on the observation that most software failures are transient in nature. The environment diversity approach requires re-executing the software in a different environment [Jalote et al, (1995)]. Environment diversity deals very effectively with Heisenbugs by exploiting their definition and nature. Adams [Adams, (1994)] has proposed restarting the system is the best approach to masking software faults. Environment diversity is a generalization of restart. Environment diversity attempts to provide a new or modified operating environment for the running software. Usually, this is done at the instance of a failure in the software. When the software fails, it is restarted in a different, error-free operating system environment state, which is achieved by some clean up operations.

Among the three fault tolerant techniques design diversity is a concept that traces back to the very early age of informatics [Avizienis, (1979); Randell, (1986)]. The approach is the most widely used and has become a reality, as witnessed by the real life systems. The currently privileged domain where design diversity is applied is the domain of safety related systems and hence is very important to study and understand. The two most well documented techniques of design diversity for tolerating software design faults are the

- **Recovery Block Scheme (RB)**
- **N-Version Programming (NVP)**
Another well-known mechanism is to use a hybrid fault tolerant scheme, which combines the features of both RB and NVP mechanism and is known as **Consensus Recovery Block Scheme (CRB)**.

**Recovery Block Scheme**: Proposed by Randell, (1975), requires *n* alternatives of a program and a testing segment called acceptance test (AT). Whenever an alternative fails, the testing segment activates the succeeding alternative. The function of the testing segment is to ensure that the operation performed by an alternative is correct. If the output of the alternative is incorrect, then the testing segment recovers the initial state and activates the next alternative.

![Recovery Block Scheme](image)

**Figure 1.1.7: Recovery Block Scheme**

**N-Version Programming Scheme**: proposed by Chen and Avizienis, (1978), a number (*N* ≥ 2) of independently coded versions (alternatives) for a given program are run concurrently and the results are compared. A voting scheme recognizes the majority as “correct output”. N-version programming is used in space shuttle avionics. A specific limitation of the N-version programming is the requirement of *N* computers that are hardware independent and able to communicate very efficiently to compare the outputs. Both N-version programming and recovery block scheme achieve fault tolerance by increasing redundancies within the software. Fault-tolerant software systems are usually developed by integrating software components.
Consensus recovery block scheme given by Scott et al (1987) incorporates both the techniques (recovery block and N-version scheme).

**Consensus Recovery Block (CRB):** [Scott et al, (1987)] is an attempt to combine the techniques used in the recovery block and NVP. It is claimed that the CRB technique reduces the importance of the acceptance test used in the recovery block and is able to handle the case where NVP would not be appropriate since there are multiple correct outputs. The CRB requires design and implementation of N alternatives of the algorithm, which are ranked (as in the recovery block) in the order of service and reliance. On invocation, all alternatives re-executed and their results submitted to an adjudicator, i.e. a voter (as used in NVP). The CRB compares pairs of results for compatibility. If two results are the same then the result is used as the output. If no pair can be found then the results of the alternatives with the highest ranking are submitted to an acceptance test. If this fails then the next alternatives is selected. This continues until all alternatives are exhausted or one passes the acceptance test. Scott et al, (1987) developed reliability models for the RB, NVP and the CRB. In comparison, the CRB is shown to be superior to the other two. However, the CRB is largely based on the assumption that there are no common faults between the variants.
Fault tolerance in software enhances the overall reliability of the system but incurs huge cost due to redundancy in software and hardware component. Therefore, developers must carry a trade-off between cost of fault tolerance against cost of system failure. Many researchers in the field of software reliability engineering have done excellent research to study the fault tolerant systems in many ways. Several optimization models have been formulated in the literature for the optimal selection of the software components [Belli and Jedrzejowicz, (1990); Ashrafi and Berman, (1992); Berman and Ashrafi, (1993); Ashrafi et al, (1994); Kumar, (1998); Berman and Kumar, (1999); Kapur et al, (2003)]. The models used basic information on components reliability and cost and allow the trade-off between two factors. Only a few authors have formulated optimal component selection model for a fault tolerant modular software system under consensus recovery block scheme. In our study, we have formulated several optimization models for component selection in component based software design incorporating various aspects such as execution time of components, build-or-buy strategy and mandatory redundancy for critical modules, for fault tolerant modular software system under consensus recovery block scheme.

1.1.8 Component Based Software Engineering (CBSE)

The extensive use of software has placed new expectations on software industry and there is an ever going push towards component based software development. The simplest definition of CBSE can be found in [Ning, (1997)] as “the process of assembling software systems from components”. A more detailed definition is [Grundy, (1999)] “building applications from discrete, inter-related software components, often dynamically plugged into running applications and reconfigured by end users or other
A component is “an encapsulated, distributable, and executable piece of software that provides and receives services through well-defined interfaces.” [Ning, (1999)]. Component based software engineering is an approach that aims to move the software industry away from developing each system from scratch. A component based software system is usually built up of multiple software modules such that each module is more manageable. Modules design is identified during the development process and integrated later to form a complete system. The modular design is even further structured into smaller components such that these small components can be developed independently and later linked to develop a full functionality. The developers choose different options for the development of these small independent components such as:

- Choosing from already available commercial off-the-shelf (COTS) components
- In-house development
- Modifying the functioning of some existing components (software reuse).

1.1.8.1 Commercial off-the Shelf Component

In the last several years, the world of software-system development has been revolutionized by commercial off-the-shelf (COTS) software products. A COTS component is a piece of software that can be reused in software projects to become a part of other software products [Torchiano and Morisio, (2004)].

**COTS – CeBase Definition**

A software product: developed by a third party (who controls its ongoing support and evolution); bought, licensed, or acquired for the purposes of integration into a larger system as an integral part, i.e. that will be delivered as part of the system to the customer of that system; which might or might not allow modification at the source code level, but may include mechanisms for customization; and is bought and used by a significant number of systems developers.

The use of the Commercially-of-the-Shelf (COTS) products as units of large systems is becoming popular. Shrinking budgets, rapid advancement of COTS development and the increasing demands of large systems are all driving the adoption of COTS development approach. A COTS component is defined as an independent unit that provides a rather than building the whole system from scratch, a new system can be assembled and constructed by using existing, market proven
and vendor supported COTS products. For example, accounts receivables and inventory control COTS components can be purchased and integrated into an accounting system. The use of COTS products, as opposed to development of products in-house, has several potential advantages such as lower purchasing cost due to the mass production, lower development effort as COTS products are ready-made, high maturity as COTS products go through multiple releases starting from beta versions to final versions, and rich functionality as COTS products cover a wide spectrum of users with different needs [Vigder et al, (1996); Voas, (1998)]. On the other hand, several challenges are encountered when using COTS products, including the lack of control over COTS development and evolution [Boehm and Abts, (1999)], and COTS’ inability to meet specific requirements in different projects [Alves, (2003)]. Table 1.1.2 summarises a number of the relative advantages and disadvantages of COTS solutions; these indicate that COTS integration differs significantly from traditional software development and requires significantly different approaches to its management [Boehm and Abts, (1999)].

Table 1.1.2: COTS Advantages and Disadvantages

| Advantages                                                                 | Disadvantages                                                                                      |
|                                                                           |                                                                                                   |
| Immediately available, earlier payback                                  | Licensing, intellectual property procurement delays                                                |
| Avoids expensive development                                              | Up front license fees                                                                             |
| Avoids expensive maintenance                                              | Recurring maintenance fees                                                                       |
| Predictable, confirmable license fees and performance                     | Reliability often unknown or inadequate; scale difficult to change                                |
| Rich functionality                                                        | Too-rich functionality compromises usability, performance                                           |
|                                | Constraints on functionality, efficiency                                                          |
| Frequent upgrades often anticipate                                        | No control over upgrades and maintenance                                                          |
|                               |                                                                                                   |
| Dedicated support organization                                           | Dependence on vendor                                                                             |
| Hardware/software independence                                            | Integration not always trivial; incompatibilities among vendors                                  |
| Tracks technology trends                                                  | Synchronizing multiple-vendor upgrades                                                             |

[Source: Boehm and Abts, (1999)]

In order to cope with such challenges, suitable methods and techniques should be used to evaluate existing COTS products, after which the most appropriate COTS is selected. Inappropriate COTS selection can result in serious consequences leading to system
failure—for example, the London Ambulance Service fiasco in 1992, in which the system descended into chaos and reverted back to manual operation partly because of inappropriate COTS selection [Maiden and Ncube, (1998)].

The use of COTS products has been recognized as a trend and it has become an economic necessity [Tran and Lui, (1997)]. The selection of the right COTS component is often a non-trivial task and requires careful consideration of multiple criteria and careful balancing between application requirements, technical characteristics and financial issues. Many authors have proposed different methods for selection of COTS component for development of component based software system. One of the first proposals was given by Kontio et al in (1995) who proposed the OTSO (Off-The-Shelf Option) approach for COTS selection in 1995. The authors developed a method that addresses the selection process of packaged, reusable off-the-shelf software. The method, called Off-the-Shelf-Option (OTSO), supports the search, evaluation and selection of software components. The overall phases of COTS software selection are presented in figure 1.1.10

The horizontal axis in figure 1.1.10 represents the progress of the evaluation (i.e. time) and vertical axis the number of alternatives considered at each phase. Starting by the search phase, the number of possible alternatives may grow quite rapidly. The most potential candidates will need to be sorted out (screening) to pick the ones that can be evaluated in more detail with the resources available. Detailed evaluation of a limited
number of alternatives determines how well each of the alternatives meets the evaluation criteria. These results are systematically documented. During the analysis phase a decision is taken for the selection of COTS component which has deployed into the final system. Multi-criteria decision making techniques help to select the best component for the system. Finally, in order to improve the selection process and to provide feedback on potential further reuse of the component, it is necessary to access the success of the reuse component used in a project.


1.1.8.2 In-house Built Components

The context and size of the software varies in different conditions, and is influenced by the programming languages used and the system and application architectures. The trend is towards assembling software from the existing software components developed using in-house resources and from the vendor supplied libraries and components. If a component is not available in the market or also if it cannot be reused then the decision is to build that component in-house from scratch. Also, sometimes it may happen that building a component in-house is economical as compare to buying it from the market. In this case as well, the right decision is to build that component. Reusing and
modifying the code of the existing components also come under the category of in-
house built component.

1.1.8.3 Software Reuse

Software systems have become large and large and more and more new methodologies
emerged to adjust to the changing needs. With the object oriented systems software
development has seen a new revolution in the development of applications. With OOPS
many of the components written in one application are being reused in the other
applications with similar functional requirement. This led to increasing the software
development performance and thereby increasing the ‘Business’ performance by
developing software “faster, better and cheaper”. Faster - as software has to meet
dynamic market requirements along with facing intense market competition. Better, as
software has to serve the requirements of the while processing and even with the few
failures. Cheaper, because then software has to be less expensive to produce and
maintain [Jacobson et al, (1997)]. Furthermore, software reuse is recognized as one of
the most efficient means for reducing the number of bugs [Alves and Finkelstein,
(2003)] and [Basili et al, (1996)] if development costs have to be kept at a
comparatively low level. Software reuse has led to development of components for
addressing some standard functionality and there by business can choose to use them
directly to solve their business need. The growing popularity and availability of
component-based software technologies is fueling a change in the habits and
expectations of millions of programmers. New application development tools and
technologies have made “Components” the key to reusing larger grained “objects” to
build applications rapidly. Examples of these technologies include Microsoft Visual
Basic, ActiveX, Sun Java, and OMG’s Corba etc. Both COTS and in-house
components can be reused and modified to get a new software product.

Literature lists several factors that drive the popularity of CBSE. These include the
increasing industrial competition for the delivery and provision of high quality and
reliable software within short time-frames, as well as the demand for larger and more
complex projects, which often cannot be accomplished by a single software development organization that has to start from scratch [Alves and Finkelstein, (2003)]. And finally, CBSE is stimulated by a growing number of technologies such as CORBA, JavaBeans/EJB, DCOM/ActiveX or .Net as well as by the availability of various component configuration and deployment tools [Andrews et al, (2005)].

1.1.9 System Design phase for Component Based Software Engineering (CBSE)

System design is concerned with how the system functionality is to be provided by the different components of the system. These components may be either COTS components or in-house developed components that may be either developed from scratch (if not at all available in the market) or it can be developed by modifying and reusing the existing components.

![Design Phase of CBSE Diagram]

[Source: Sommerville, (2001)]

Figure 1.1.11: Design Phase of CBSE

The activities involved in the design process are:

1. **Partition Functional Requirements**: The requirements are analyzed and collected into related groups. There are usually several possible options for partition and a number of alternatives may be produced at this phase of the process.

2. **Identify Modules**: Different modules that can individually or collectively meet the requirements are identified. Independent components are assembled to form a module. Groups of requirements are usually related to the modules, so this
activity and requirements portioning may be amalgamated. However, the module identification may also be influenced by other organizational or environmental factors.

3. **Assign requirement to modules**: the requirements are assigned to modules. In principle, this should be straightforward if the requirements portioning is used to derive the module identification. Sometimes there is not a clean match between requirements partitions and modules identified. Limitations of externally purchased components for modules may mean that requirements have to be modified. If the requirements do not match with the COTS product, then that component can be built in-house from the scratch or sometimes the existing software product can be modified and reused.

4. **Specify module functionality**: the specific functions provided by each module are specified. This may be seen as part of system design phase, or if the module is a software system, part of the requirements specification activity for that system. Relationships between modules should also be identified at this stage.

5. **Define module interfaces**: this involves defining the interfaces that are provided and required by each module. Once these interfaces have been agreed, parallel development of the modules becomes possible.

As the double ended arrows in the figure 1.1.11 imply, there is a great deal of feedback and iteration from one stage to another in the design process. As problems and questions arise, rework of earlier stages is often necessary.

### 1.1.10 Component Based Software Structure for Fault Tolerant Systems

With the growing emphasis on CBSE, an increasing number of organizations are developing and using software not just as all-inclusive applications, as in the past, but also as component parts of larger applications. Component based software system focuses on the decomposition of a software system into functional or logical components with well defined interfaces. It allows a software system to be developed
using appropriate software components. This approach has been proven to facilitate shorter software systems development time and better system maintainability [Szyperski et al, (2004)]. The motivation for the use of component-based approach for quantitative assessment of software systems includes the following:

- Developing techniques to analyze the reliability and performance of applications built from reusable and COTS software components.
- Understanding how the system reliability/performance depend on its component reliabilities/ performance and their interactions.
- Studying the sensitivity of the application reliability to reliabilities of components and interfaces.
- Guiding the process of identifying critical components and interfaces.
- Developing techniques for quantitative analysis that are applicable throughout the software life cycle.

Therefore, optimal selection of components for a fault tolerant modular software system is a complex optimization problem of software development process using CBSE approach. Two different types of software structures are discussed (see figure 1.1.12 and 1.1.13).

Large software system has modular structure to perform set of functions with different modules having different alternatives for each module. A schematic representation of the software system is given in figure 1.1.12. Components (alternatives) are to be selected for modules to maximize the system reliability by simultaneously minimizing the cost/ execution time. These components may be the COTS components or in-house built components (reused or new ones). The selected components in the software structure are highlighted with pink colour and the components which are not selected are shown in blue colour. The frequency with which the functions are used is not same for all of them and not all the modules are called during the execution of a function the software has in its menu.
Figure 1.1.12: Structure of Software with Alternatives

Further the above approach can be broken down into various versions of alternative given in figure 1.1.13. Different versions for each alternative are available in the market. Only one version is to be selected for each alternative of a module.
The optimization models for component selection developed in this thesis are based on either of the two software structures. A system can be made fault tolerant by adding redundancy. Therefore, in both the cases more than one alternative can be selected for a module and hence redundancy is allowed.
1.1.11 Decision Making for CBSE

Decision-making in today’s fast paced, competitive world has become a complex task. High cost of technology, resources, competitive pressures and other factors greatly increase the difficulty of decision making. In component based software development process, one of the crucial and specific activities is component selection [Cortellessa, (2008)]. The decision to choose best set of (COTS and in-house, either new or reuse) components for a CBSE is not an easy task, it involves a trade-off between quality and cost along with many other concerns. Today market is overloaded with multiple software components for almost every single purpose. The buying of software and especially COTS software components is regarded as something that needs to be meticulously decided. Manager cannot take this decision based on his intuition, or the past experiences. He has to rely upon the scientific way of decision making. Mathematical modeling and optimization in the scientific field of Operational Research offers promising solution to this decision making problem.

Operational Research (O.R.) is the use of advanced analytical techniques to improve decision making. It makes sense to make the best use of available resources. Today’s global markets and instant communications mean that customers expect high quality products and services when they need them, where they need them. Organizations need to provide these products and services as effectively and efficiently possible. This requires careful planning and analysis—the hallmark of good O.R. The foundation of operational research is mathematical modeling.

1.1.12 Mathematical Modelling

The branch of mathematics that consists of developing a mathematical model that is used to solve a real world problem is referred to as Mathematical Modeling. A mathematical model is an abstraction of the assumed real world, expresses in an amenable manner as mathematical functions that represent the behavior of the assumed systems [Taha, (2006)]. By using mathematical modeling techniques to
analyze complex situations, operations research gives executives the power to make more effective decisions and build more productive systems based on:

- Information and data analysis
- Consideration of all available alternatives
- Careful predictions of outcomes and estimation
- The latest decision tools and techniques

Optimization is a powerful tool of operational research and is regarded as an important methodology of conceptualization and analysis. A model can be developed with respect to a system to measure some particular quantity of interest such as cost model or a profit model or it may represent the assumed system as a whole used to optimize the system performance, the *optimization model*. The optimization models developed for the engineering and business professional allow them to choose the best course of action and experiment with the various possible alternative decisions.

![Ingredients of Mathematical Optimization](image)

**[Source: Kallrath, (2003)]**

**Figure 1.1.14: Ingredients of Mathematical Optimization**

The components of an optimization model include

- Objective function(s)
- Decision variables
- Constraints
Objective Function: The objective function(s) in optimization models measures the performance of the system. The performance is measured in terms of maximization (i.e. profit, reliability) or minimization (i.e. loss, cost). In many situations, organization may have more than one objective and is known as multi objective decision-making problem. In this thesis, we have formulated various multi-objective optimization models for selection of components in design of fault tolerant modular software system. Various multi-objective optimization models are formulated where the two objectives are:

- Maximization of system reliability and minimization of the overall system cost
- Maximization of system reliability and minimization of execution time of the system

Decision variables: The variables whose values are under our control and influence the performance of the system are called decision variables. Decision variables can be continuous, discrete or Boolean. The decision variables in multi-objective optimization models for component selection are Boolean whose respective values tell whether a component is selected for a particular module in design of a fault tolerant modular software system.

Constraints: In most situations, only certain values of decision variables are possible. Restrictions on the values of decision variables are called constraints. For example, again in case of optimal component selection model, delivery time can be considered as a constraint. We want a set of components to be selected for a software system under the desired delivery time.

In optimization models values of the decision variables produce the optimal result.

Mathematical programming is powerful optimization technique which may serve the purpose of solving optimization problems effectively and efficiently. The major steps required to perform for solving a typical optimization are as follows:
1. Formulate the problem
2. Observe the system
3. Formulate a mathematical model
4. Verify the model
5. Select the best alternative
6. Present the results of the analysis
7. Implement and evaluate

1.1.13 Mathematical programming form for Optimization Problems

In mathematics, the term optimization, or mathematical programming refers to the study of problems in which one seeks to minimize or maximizes single/multiple functions by systematically choosing the values of real or integer variables or binary form within an allowed set, depending upon the real life/problem based situations. The type of mathematical programming which have been used for applications purpose while designing optimization models [For details, refer (appendix A)].

There are two categories of optimization models.

- Crisp optimization models
- Fuzzy optimization models

1.1.13.1 Crisp Optimization Models

For the mathematical programming problems in the crisp scenario, the aim is to maximize (and/or) minimize the objective function under set of constraints which takes the exact or deterministic values of the parameters of the optimization models. In case of software development using CBSE approach, information on cost, reliability and delivery time is considered to be deterministic in nature. The model is then solved using exact values of the parameters and hence called crisp optimization models.

1.1.13.2 Fuzzy Optimization Models

Classical mathematical programming models require well-defined criterion and activity constant coefficients, resource, requirement and structural conditional constants together
with well-defined inequalities. The coefficient and conditional constants are calculated based on the system information where it is assumed that the system behavior and environment are deterministic in nature, which is not true in general. In most real life problems it is observed that some (or all in some cases) of the model constants can only be computed roughly due to their dependence on various non-deterministic factors. Besides this there are various other sources that bring uncertainty in the computation such as system complexity, subject’s awareness, communication of and thinking about uncertainty, soft information, ambiguous statements, intended flexibility and economics of information, etc. Hence the optimal solution of the problem so obtained is not actually representative of the complete system information and can provide us with the incorrect picture of the system. In such cases defining the problem under fuzzy environment offer the opportunity to model subjective imagination of the decision maker as precisely as a decision maker will be able to describe it. Optimization under fuzzy environment is gaining even more importance in the era of globalization due to some additional sources contributing to bringing uncertainties in the problem definitions. They include increasing uncertainties in quantifying the process parameters, stiff competition, shorter life cycles of products, increased influence of small changes in the process variables on the system performance, etc. Some time the decision makers themselves specify ambiguous statements for objectives and constraints in order to have some tolerance on the optimal values due to competitive considerations.

The direct application of techniques of mathematical programming to solve fuzzy optimization problems doesn’t serve the purpose as it provide no mechanism to quantify the uncertainties. The fuzzy set concept and fuzzy optimization techniques can be used efficiently in such situations to defuzzify the fuzzy parameters, constraints and objectives and formulating an equivalent crisp problem which can further be solved using the mathematical programming techniques [Zimmermann, (1991); Bector and Chandra, (2005); Lee, (2005)]. Even though the fuzzy optimization procedure is highly subjective to the problem solver due to the subjectivity of the defuzzification techniques it is preferred as it provides flexibility in terms of the alternative courses of action to the decision maker. Some preliminary concepts of fuzzy set theory and fuzzy optimization are discussed in Appendix B.
In the thesis we have formulated crisp and fuzzy optimization models for component selection for a fault tolerant modular software system. *Zero one* integer programming problems have been formulated in which decision variables are allowed to take a value of one or zero depending on whether the software component is selected for a module or not.

### 1.1.14 Thesis Contribution

This thesis endeavors to apply techniques of Operational Research to modeling and optimization in component selection for a Component based software engineering for fault tolerant modular software systems.

With increasing proliferation and diversity of COTS components selection of components for designing a software system optimally is a critical and complex optimization domain of CBSE. The component based software development approach has been proven to facilitate shorter software systems development time and better system maintainability. Therefore the question arises *“What mix of COTS components must be chosen in designing a fault tolerant software system so as to have a highly reliable system subject to various resource constraints?”* This problem is addressed in chapter 2 and 3 which focuses on formulation of multi-objective crisp and fuzzy optimization models for selection of COTS components in designing a fault tolerant software system.

While developing software, components can be both bought as COTS products, and probably adapted to work in the software system, or they can be developed in-house. This decision is known as “build-or- buy” decision. The question arises *“How to optimally select the components for a software system using build-or-buy strategy?”* The problem of selecting components using build-or-buy approach is discussed in chapter 4. Fuzzy multi-objective optimization models are formulated for the same.

While developing critical software systems the developer must identify the critical modules of the software and make sure that the probability of failure of these critical modules must be negligible. This can be done by adding mandatory redundancy to
critical modules. Now the question arises “How to create mandatory redundancy for critical modules of the system while designing a fault tolerant modular software system”. The thesis contributes in this area as well and the issue of criticality of modules is discussed in chapter 5.

1.2 REVIEW OF LITERATURE ON OPTIMAL COMPONENT SELECTION PROBLEM

Several optimization models have been proposed in the literature for the optimal selection of the software components for the development of safe and reliable software systems. The models used basic information on components reliability and cost and allow the trade-off between two factors. At the outset Scott et al, in (1987) examined three methods of creating fault tolerant software systems, Recovery Block Scheme, N-Version Programming, and Consensus Recovery Block. The authors presented reliability models for each technique. The models are used to show that one method, the Consensus Recovery Block, is more reliable than the other two. Consensus Recovery Block systems were more reliable than corresponding Recovery Block only when a highly reliable acceptance test was used. In N-Version Programming system does not recognize an output as correct unless it occurs more than once. McAllister and Scott in (1991) compared cost of a single version system with the three versions of fault tolerant software systems. The authors shown that in cases where failures are independent, Consensus Recovery Block followed by Recovery Block are the most cost justifiable fault tolerant techniques to be considered. Unless the voter is perfect, N-Version Programming does not compete with the other two methods.

All fault tolerance techniques provide some degree of reliability improvement. On the other hand all techniques of fault tolerance relies on design or data redundancy to mask failure or recover from the state of failure. A direct implication of building redundancy is the additional cost. As we are all aware that with the growth of complexity and requirement of software in critical systems fault tolerance can’t be avoided. Hence it is desired that the developers must weigh the cost of fault tolerance against the cost of failure. Instead of applying ad hoc methods, use of optimization techniques is always preferred in such decision-making situations. Optimization problems of optimum selection of redundant components are widely studied by many researchers in the literature [Belli and Jedrzejowicz, (1991); Ashrafi and Berman, (1992); Berman and Ashrafi, (1993); Ashrafi et al, (1994); Kumar, (1998); Berman and Kumar, (1999); Kapur et al, (2003); Cortellessa et
Different optimization models were formulated each applicable to a different software system structure, ranging from a very simple configuration to more sophisticated ones. The earliest work in this field appears to be done by Belli and Jedrzejowicz, (1991) but it gained popularity with the work of Ashrafi, (1992) and Berman, (1993).

1.2.1 Optimization Models for Selection of Programs, Considering Cost and Reliability [(Ashrafi and Berman, (1992)]

The authors presented two models which address the tradeoff between reliability and cost. They apply to large software packages that consist of several programs. According to Pressman, (2005) a program is defined as a set of complete machine instructions (operations with operands) which upon execution, perform one major function as required by the user. The models can be used as decision support tools for organizations that are in the process of purchasing of variety of computer programs in order to meet the needs of the users, e.g. operations people need software packages to perform functions such as scheduling, inventory control and purchase orders. While the main consideration is to attain high average reliability for the software package, management has to consider both the relative importance of each program in terms of the frequency of the usage of their corresponding function. Programs can be purchased from software development companies. Several programs are usually available for each function. Each program has a known market cost and an estimated reliability. It can noted here that the assumption of using the ready programs available in the market to make the software package implies the use of COTS (Commercial-off-the-shelf) program. Hence the models are applicable only those software packages that are designed using COTS products. The authors have formulated to types of models one which does not consider maintain redundancy for performing each function and the other which maintains redundancy under budget limits. Models are based on the following assumptions.

Assumptions

1. Each program has a distinct, known reliability and cost.
2. The budget is limited.
3. Usage frequency for each function is known, viz, provided by the user.
Notations

\( K \)  
Number of functions the software package is required to perform

\( F_k \)  
Frequency of use of function \( k \), \( k = 1,2,\ldots, K \)

\( m_k \)  
Number of programs available for function \( k \)

\( R_{kj} \)  
Reliability of program \( j \), which performs function \( k \)

\( X_{kj} \)  
Indicator is 
\[
X_{kj} = \begin{cases} 
1 & \text{program is selected to perform function} \\
0 & \text{otherwise} 
\end{cases}
\]

\( \bar{R} \)  
Average reliability of the software package

\( c_{kj} \)  
Cost of developing program \( j \) that performs function \( k \)

\( B \)  
Available budget

1.2.1.1 Selecting an Optimal Program for each Function (without Redundancy)

The software package performs several functions but due to financial limitations and/or critical nature of the functions, keeping multiple programs that are functionally equivalent is not possible. The objective is to select one program for each function such that the average reliability is maximized and cost of purchasing programs remains within the budget.

Problem (1.2.P1)

Maximize \( \bar{R} = \sum_{k=1}^{K} F_k R_k \)  
\( \quad \ldots(1.2.1) \)

Subject to

\[
\sum_{j=1}^{m_k} X_{kj} = 1 ; \ k = 1,\ldots, K
\]  
\( \quad \ldots(1.2.2) \)

\[
\sum_{k=1}^{K} \sum_{j=1}^{m_k} X_{kj} c_{kj} \leq B
\]  
\( \quad \ldots(1.2.3) \)

\[
X_{kj} = 0,1 ; \ k = 1,2,\ldots, K ; \ j = 1,2,\ldots, m_k
\]
\[ R_k = \sum_{j=1}^{m_k} x_{kj} R_{kj} \]  \hspace{1cm} \text{...(1.2.4)}

The objective function of the above problem reflects that the average reliability of the software package is maximized which is a weighted sum of the reliability of the \( K \) function; reliability of each program is multiplied by the usage frequency of the corresponding function. The constraint set ensures that exactly one program is selected for each function and the total expenditure does not exceed the budget. Since the functions are required for entirely different purposes they should be considered not as a series of functions but as a set of s-independent functions.

Maximizing the average reliability \( \bar{R} \), is equivalent to minimizing the average failure rate \( \bar{Z} \),

\[ \bar{Z} \equiv (N / H) \sum_{k=1}^{K} F_k (1 - R_k) \]  \hspace{1cm} \text{...(1.2.5)}

where

- \( H \) : Operation time
- \( N \) : Number of runs during \( H \)

\[ 1 - R_k \] : Probability of failure for function \( k \)

Using \( \sum_{k=1}^{K} F_k = 1 \) equation (1.2.5) can be rewritten as

\[ \bar{Z} = (N / H) \left( 1 - \sum_{k=1}^{K} F_k R_k \right) = (N / H)(1 - \bar{R}) \]

The problem (1.2.P1) is an integer-programming problem and can be solved using software packages such as LINGO, LINDO, Mathematica etc. The problem has sum of \( m_k; k = 1, 2, ..., K \) variables and \( K+1 \) constraints. The authors have also proposed a Lagrangian Relaxation algorithm to solve the problem if \( K \) and \( m_k \) are large. However, the professional versions of these software are available which can solve problems of very high dimensions.
1.2.1.2 Selecting Optimal Set of Programs for each Function (with Redundancy)

This model costs more than model (1.2.1). The objective is to determine the optimal set of programs for each function, allowing redundancy, so as to maximize the average reliability of the software package while remaining within the budget.

Problem (1.2.P2)

Maximize $\bar{R} = \sum_{k=1}^{K} F_k R_k$

Subject to

$$\sum_{j=1}^{m_k} X_{kj} \geq 1, \quad k = 1, \ldots, K \quad \ldots(1.2.6)$$

$$\sum_{k=1}^{K} \sum_{j=1}^{m_k} X_{kj} C_{kj} \leq B$$

$$X_{kj} = 0, 1 \quad ; \quad k = 1, 2, \ldots, K \quad ; \quad j = 1, 2, \ldots, m_k$$

$$R_k = 1 - \prod_{j=1}^{m_k} \left(1 - R_{kj}\right)^{X_{kj}} \quad \ldots(1.2.7)$$

In contrast to (1.2.2), (1.2.6) allows selection of more than one program for any particular function. Here it is assumed that the cost of performing the acceptance test for each program is negligible compared to the purchase cost. A failure state occurs only when all alternative programs for any function gives an incorrect output. Thus $R_k$ is the probability that at least one of the programs selected for the function $k$ is working. This probability is hence same as that for a parallel system. The problem (1.2.P2) can be rewritten as a minimization of failure rate using (1.2.7) with the objective function

$$\text{Minimize} \quad \sum_{k=1}^{K} F_k \prod_{j=1}^{m_k} \left(1 - R_{kj}\right)^{X_{kj}} \quad \ldots(1.2.8)$$

Like problem (1.2.P1), (1.2.P2) can also be solved using any integer-programming package. However the authors have proposed a dynamic programming algorithm to
solve the problem as due to its nonlinear nature it can’t be solved with Langrangian Relaxation algorithm.

1.2.2 Optimization Models for Reliability of Modular Software Systems [Berman and Ashrafi, (1993)]

Considering the concept of COTS in a software development and the availability of mathematical models to access module reliability, it is now possible to have information on module reliability and cost. In their previous work [Ashrafi and Berman, (1992)] authors used optimization models to determine the redundancy level of a software package consisting of several independent functions where each function is performed by a program with known reliability and cost. In this work, however, they break down this approach one step further and deals with software systems consisting of one or more programs where each program consists of series of modules, which upon sequential execution will perform a function. Four models are presented, each applicable to a different software system structure. Assumptions to the models are as follows.

Assumptions

1. Module programming is used for software development
2. Module versions are developed independently, and their reliabilities and costs can be estimated.
3. There is a specified budget for the software system.

Notations

\[ K \] number of functions the software package is required to perform
\[ n \] number of modules in the software
\[ F_k \] frequency of use of function \( k, k = 1,2,\ldots, K \)
\[ m_i \] number of versions available for module \( i \)
\[ R_{ij} \] reliability of version \( j \), for module \( i \)
\[ X_{ij} \] indicator is \( \begin{cases} 1 & \text{Version } j \text{ is selected for module } i \\ 0 & \text{otherwise} \end{cases} \)
\( R_i \)  
estimated reliability of module \( i \)

\( \bar{R} \)  
estimated reliability of the software

\( c_{ij} \)  
cost of developing version \( j \) for module \( i \)

\( B \)  
available budget

\( S_k \)  
set of modules corresponding to program \( k \)

1.2.2.1 Selecting the Optimal Set of Modules for One Function System (Without Redundancy)

Software system consists of a single program performing one major function. The program is comprised of a set of modules, which are executed sequentially. The model developed for the single program such that the reliability is maximized while meeting the constraint that the overall development cost remains within budget.

Problem (1.2.P3)

Maximize  
\[
R = \prod_{i=1}^{n} R_i
\]

Subject to

\[
\sum_{j=1}^{m_i} X_{ij} = 1 \quad ; i = 1, \ldots, n \quad \ldots (1.2.9)
\]

\[
\sum_{i=1}^{n} \sum_{j=1}^{m_i} X_{ij} C_{ij} \leq B
\]

\[
X_{ij} = 0, 1 \quad ; i = 1, \ldots, n \quad ; j = 1, \ldots, m_i
\]

where

\[
R_i = \sum_{j=1}^{m_i} X_{ij} R_{ij} \quad \ldots (1.2.10)
\]

The objective function of (1.2.P3) reflects that the modules are executed sequentially. The constraints set ensures that exactly one version is selected for each module and total expenditures will not exceed \( B \). This problem is a nonlinear integer-programming problem. Authors suggested a Branch and Bound approach to solve the problem.
although it can also be solved using any nonlinear integer programming software package.

1.2.2.2 Selecting the Optimal Set of Modules for One Function Software System (With Redundancy)

When the software system perform more critical functions whose failure can be very severe. In such situations, software can be made fault-tolerant by keeping redundant versions for each module. The model is developed to determine the optimal set of modules, allowing redundancy, so as to maximize the reliability of the software system while remaining within the budget.

Problem (1.2.P4)

\[
\text{Maximize } \quad R = \prod_{i=1}^{n} R_i \\
\text{Subject to} \\
\sum_{j=1}^{m_i} X_{i,j} \geq 1 \quad ; \quad i = 1, \ldots, n \quad \ldots(1.2.11)
\]

\[
\sum_{i=1}^{n} \sum_{j=1}^{m_j} X_{i,j} C_{i,j} \leq B
\]

\[
X_{ij} = 0,1 \quad ; \quad i = 1, \ldots, n \quad ; \quad j = 1, \ldots, m_i
\]

where

\[
R_i = 1 - \prod_{j=1}^{m_i} \left(1 - R_{ij}\right)^{X_{ij}} \quad \ldots(1.2.12)
\]

The reliability of module \( i \) is defined as the probability that at least one of the \( m_i \) versions is performing correctly. The constraint set guarantees that for each module \( i \) at least one version will be selected. Again this problem is also a nonlinear integer-programming problem. Authors have suggested a Dynamic programming algorithm for solving the problem. Since this problem can be solved using software packages we avoid discussing their algorithm.
1.2.2.3 Selecting the Optimal Set of Module for a System with K Functions (Without Redundancy)

This model deals with software systems consisting of several programs each perform a specific function. Each program contains a series of modules. Programs can be called by their corresponding functions and modules can be called by any Program. To determine the optimal set of modules for the programs and not allowing redundancy, such that the reliability of the software is maximized while remaining within the budget. Due to the limitation of budget or non-critical nature of software operation redundancy of software modules are not allowed, hence the problem can be formulated as

**Problem (1.2.P5)**

Maximize \( R = \sum_{k=1}^{K} F_k \prod_{i \in S_k} R_i \)

Subject to

\[
\sum_{j=1}^{m_i} X_{i,j} = 1 \quad ; \quad i = 1, \ldots, n
\]

\[
\sum_{i=1}^{n} \sum_{j=1}^{m_i} X_{i,j} C_{i,j} \leq B
\]

\[
X_{i,j} = 0, 1 \quad ; \quad i = 1, \ldots, n \quad ; \quad j = 1, \ldots, m_i
\]

where

\[
R_i = \sum_{j=1}^{m_i} X_{i,j} R_{i,j}
\]

The objective function of the problem (1.2.P5) reflects that the modules in any program are executed sequentially each having usage frequency \( F_k \). The constraints set being same as in problem (1.2.P3). Authors suggested a Branch and Bound approach as well as the use of any nonlinear integer programming software package to solve the problem.

1.2.2.4 Selecting the Optimal Set of Modules for a System with K Functions (With Redundancy)

The problem is to be discussed now is identical to previous one except that redundancy is allowed.
Problem (1.2.P6)

Maximize \[ R = \sum_{k=1}^{K} F_k \prod_{i \in S_k} R_i \]

Subject to

\[ \sum_{j=1}^{m_j} X_{ij} \geq 1 \quad ; i = 1, \ldots, n \]

\[ \sum_{i=1}^{n} \sum_{j=1}^{m_j} X_{ij} C_{ij} \leq B \]

\[ X_{ij} = 0,1 \quad ; i = 1, \ldots, n \quad ; j = 1, \ldots, m_j \]

where

\[ R_i = 1 - \prod_{j=1}^{m_j} (1 - R_{ij})^{X_{ij}} \]

Again the problem is nonlinear integer programming problem and can be solved with the help of software packages.

1.2.3 Optimization Models for Recovery Blocks [Berman and Kumar, (1999)]

The optimization models discussed in the previous sections don’t consider any of the fault tolerance schemes such as recovery block or NVP. They merely consider the programs consisting of set of modules, which on sequential execution perform the function. Berman and Kumar, (1999) studied the problem of optimum selection of component for the recovery blocks for the first time. The author presented optimization models for a fault tolerant software system. Specifically they have formulated optimization problems for two types of recovery blocks namely — Independent and Consensus recovery block schemes.

Notations

\( n \) Number of versions in the recovery block

\( P(.) \) Probability of the event (.)

\( P_i \) Failure probability of version \( i \)
Chapter 1: Introduction

$C_i$  Cost of version $i$

t$_1$  Probability that the testing segment cannot perform successful recovery of the input state

t$_2$  Probability that the testing segment rejects a correct result

t$_3$  Probability that the testing segment accepts an incorrect result

$B$  Available budget

$R_n$  Reliability of a recovery block with $n$ versions

$RC_n$  Reliability of a consensus recovery block with $n$ versions

$Z_j$  Binary variable $= \begin{cases} 1 & \text{if version } j \text{ is included in the software} \\ 0 & \text{otherwise} \end{cases}$

$Z_{i,j}$  Binary variable $= \begin{cases} 1 & \text{if version } i \text{ is placed at position } j \text{ in the software} \\ 0 & \text{otherwise} \end{cases}$

$R_j$  Reliability for the partial solution $\{Z_1, Z_2, \ldots, Z_j\}$

**Independent Recovery Block**

Different types of errors that can result in failure of a recovery block are

1. A version produces correct result, but the testing segment labels it incorrect
2. A version produces incorrect result, but the testing segment labels it correct
3. The testing segment cannot perform successful recovery upon failure of a version

To compute the reliability of a recovery block based on the failure modes observed two types of events are defined. Let

$Y_i$  : Event that version $i$ produces a correct result and testing segment accepts it as correct

$X_i$  : Event that either version $i$ produces an incorrect result or produces a correct result and the testing segment rejects it; in both cases the testing segment performs a successful recovery of the input states.

The probabilities corresponding to the above two events are given as
Optimal Component Selection for Fault Tolerant Software Design under Consensus Recovery Block Scheme

\[ P(Y_i) = (1 - p_i)(1 - t_2) \]

\[ P(X_i) = (1 - t_i)\left[ p_i(1 - t_3) + (1 - p_i)t_2 \right] \]

Now the reliability of a recovery block scheme with a single version 1, \( R_1 \) is defined as

\[ R_1 = P(Y_1) \]

In general reliability of a recovery block with \( n \) versions is

\[ R_n = P(Y_1) + \sum_{i=2}^{n} \prod_{k=2}^{i-1} P(X_k) \left[ P(Y_i) \right] ; \quad n \geq 2 \]

...(1.2.13)

Recursively,

\[ R_n = R_{n-1} + \prod_{k=1}^{n-1} P(X_k) \left[ P(Y_n) \right] ; \quad n \geq 2 \]

...(1.2.14)

To attain the largest possible reliability of a recovery block the different versions are to be installed in the order from smallest to the largest based on failure probabilities. This have been proved by Berman and Kumar, (1999).

**Theorem 1.2.1**: For a recovery block scheme with \( n \) independent versions, the list ordered from smallest to the largest based on failure probabilities is at least as reliable as any other list of the \( n \) versions.

(Proof of the theorem is given in Appendix C)

Now the problem of maximizing reliability by choosing an optimal set of versions subject to a budget constraint is

**Problem (1.2.P7)**

Maximize \( R_n = \sum_{i=1}^{n} Z_i \left( \prod_{k=1}^{i-1} P(X_k)^{Z_k} \right) P(Y_i)^{Z_i} \)

...(1.2.15)

Subject to

\[ \sum_{i=1}^{n} C_i Z_i \leq B \]

\[ Z_i = 0,1 ; \quad i = 1, \ldots, n \]
Equation (1.2.15) defines the reliability of a recovery block scheme chosen from $n$ versions corresponding to a solution $\{Z_1, Z_2, ..., Z_j\}$. The constraint ensures the budget restriction. Berman and Kumar, (1999) developed a branch and bound algorithm to solve the problem, although they have also suggested to use any mathematical programming software package to solve the problem.

**Consensus Recovery Block**

Constitutes of a consensus recovery block are $n$ independent versions of a program, an acceptance test and voting procedure. The versions are ranked, based on their failure procedure. Upon invocation of consensus recovery block, all versions are executed simultaneously and submit their outputs to a voting procedure. If the outputs of two or more versions are in agreement that output is designated as correct. Otherwise the next stage is entered. At this stage the best version is examined by an acceptance test. If the output is accepted, it is treated as correct. However, if the output is not accepted, the next best version is subject to testing. This process continues until acceptable output is found or all the $n$ outputs are exhausted.

Define the following probabilities

\[
P(G_n) = P(\text{2 or more outputs agree})
\]

\[
P(G_c) = P(\text{recurring output is correct})
\]

\[
P(D_n) = P(\text{all the outputs are different})
\]

Reliability of consensus recovery block is hence given as

\[
RC_n = P(G_n)P(G_c) + P(D_n)R_n = (1 - P(D_n))P(G_c) + P(D_n)R_n \quad \ldots (1.2.16)
\]

where $R_n$ is the reliability of a recovery block scheme with $n$ versions as given by (1.2.13) or (1.2.14). For consensus recovery block $t_i=0$ hence the probabilities of events $Y_i$ and $X_i$ are
\[ P(Y_i) = (1 - p_i)(1 - t_2) \]  
\[ P(X_i) = p_i(1 - t_3) + (1 - p_i)t_2 \]

Here, \( P(D_n) \) is the probability that at least \((n-1)\) versions of the \(n\) versions fail.

Therefore,

\[
P(D_n) = \sum_{i=1}^{n} \frac{1}{p_i} \left[ \prod_{k=1}^{i} p_k \right] (1 - p_i) + \prod_{i=1}^{n} p_i = P(D_{n-1})p_n + \left( \prod_{i=1}^{n-1} p_i \right)(1 - p_n) \]

To simplify it is assumed that \( P(G_c) = 1 \).

Hence

\[
RC_n = 1 - P(D_n) + P(D_n)R_n = 1 + P(D_n)[R_n - 1] \\
= 1 + \left[ \sum_{i=1}^{n} \frac{1}{p_i} \left[ \prod_{k=1}^{i} p_k \right] (1 - p_i) + \prod_{i=1}^{n} p_i \right] \sum_{i=1}^{n-1} \left[ \prod_{k=1}^{i-1} p(X_k) \right] P(Y_i) - 1 \]

Improvement in reliability of consensus recovery block scheme over recovery block is

\[
RC_n - R_n = 1 - P(D_n) + P(D_n)R_n - R_n \\
= (1 - P(D_n))(1 - R_n) \]

Similar to the case of independent recovery blocks reliability of a consensus recovery block is largest if the \(n\) versions are arranged in the order of their reliability. It is proved in the following theorem.

**Theorem 1.2.2**: For a consensus recovery block scheme with \(n\) versions, the list of versions ordered from smallest to largest based on failure probability is more reliable than any other list of versions.

This theorem can be proved on the similar lines of the theorem established for the case of independent recovery blocks and is discussed in Appendix C.
Now the problem of maximizing reliability by choosing an optimal set of versions subject to a budget constraint is

**Problem (1.2.P8)**

\[
\text{Maximize } \quad RC_n = 1 + \sum_{i=1}^{n} \frac{1}{p_i^{z_i}} \left( \prod_{k=1}^{z_i} p_i^{z_i} \right) \left( 1 - p_i^{z_i} \right) + \sum_{i=1}^{n} p_i^{z_i} \left[ \sum_{i=1}^{n} \left( \prod_{k=1}^{z_i} P(X_k)^{z_i} \right) P(Y_i)^{z_i} - 1 \right]
\]

Subject to

\[
\sum_{i=1}^{n} C_i Z_i \leq B
\]

\[
Z_i = 0, 1 \quad ; i = 1, \ldots, n
\]

Equation (1.2.22) defines the reliability of a consensus recovery block scheme chosen from \( n \) versions corresponding to a solution \( \{Z_1, Z_2, \ldots, Z_j\} \). The constraint ensures the budget restriction. Berman and Kumar, (1999) developed branch and bound algorithm to solve the problem, although they have also suggested to use any mathematical programming software package to solve the problem.

### 1.2.4 Optimal Reliability Allocation Problem for a Modular Software System

[Kapur *et al.*, (2003)]

The optimal component selection problem addressed due to Kapur *et al.*, (2003) considers software built by assembling COTS component performing multiple functions. Each function is performed by calling a set of modules. Modules can be assembled in a recovery block scheme to provide the fault tolerance. Again for each alternative version multiple choices are available from the supplier with distinct reliability and cost. The version for any alternative choice having higher reliability has higher cost. Two models are formulated for weighted maximization of system reliability, weights being decided with respect to access frequency of functions within the available budget. Each module is comprised of set of COTS alternative that are available in the market.
Optimal Component Selection for Fault Tolerant Software Design under Consensus Recovery Block Scheme

Notations

$\bar{R}$  Estimated reliability of the software

$F_i$  Frequency of use of function $l$, $l = 1,2,\ldots,L$

$S_l$  Set of modules required for function $l$

$R_i$  Estimated reliability of module $i$

$L$  Number of functions the software package is required to perform

$N$  Number of modules in the software

$m_i$  Number of alternatives available for module $i$

$V_{ij}$  Number of versions available for module $i$

$R_y$  Reliability of alternative $j$, for module $i$

$R_{yk}$  Reliability of version $k$ of alternative $j$, for module $i$

$X_{yk}$  Indicator is $1$ Version $k$ of alternative $j$ is selected for module $i$

$0$ otherwise

$C_{yk}$  Cost of version $k$ of alternative $j$ for module $i$

$B$  Available budget

$M$  Large number greater than one

$Y_i$  Indicator is $1$ if constraint $i$ is inactive

$0$ otherwise

$Z$  Number of alternatives compatible for module with respect to another module.

Assumptions

1. Codes written for integration of modules don’t contain any bug.

2. Other than available cost-reliability versions of an alternative, existence of virtual versions is assumed having negligible reliability of 0.001 and zero cost. Existence of virtual versions allows no redundancy in case of insufficient budget. These
components are denoted by index one in the third subscript of \( X_{ijk}, c_{ijk} \) and \( r_{ijk} \); e.g. \( r_{ij} \) is reliability of first version of alternatives \( j \) for module \( i \), having the above property.

### Optimization Model 1

The optimization problem of model 1 is

**Problem (1.2.P9)**

Maximize \( \bar{R} = \sum_{l=1}^{L} F_l \prod_{i \in s_l} R_i \) \hspace{1cm} (1.2.23)

Subject to

\[
\sum_{j=1}^{m_i} \sum_{k=1}^{n} v_{ij} c_{ijk} X_{ijk} \leq B \hspace{1cm} (1.2.24)
\]

\[
R_i = 1 - \prod_{j=1}^{m_i} \left( 1 - R_{ij} \right) \hspace{1cm} i = 1, 2, \ldots, n \hspace{1cm} (1.2.25)
\]

\[
R_{ij} = \sum_{k=1}^{m_i} X_{ijk} R_{ijk} \hspace{1cm} i = 1, 2, \ldots, n \hspace{1cm} j = 1, 2, \ldots, m_i \hspace{1cm} (1.2.26)
\]

\[
\sum_{k=1}^{m_i} X_{ijk} = 1 \hspace{1cm} i = 1, 2, \ldots, n \hspace{1cm} j = 1, 2, \ldots, m_i \hspace{1cm} (1.2.27)
\]

\[
R_i > 1 - \prod_{j=1}^{m_i} (1 - R_{ij}) \hspace{1cm} (1.2.28)
\]

Objective function maximizes software reliability through a weighted function of functional usage frequencies. Reliability of functions that are performed more frequently, consequently the modules that are invoked more frequently during use are given higher weights. Constraint (1.2.24) ensures the budget restriction. As it is assumed that the exception raising and control transfer programs work perfectly, a module fails if all attached alternatives fail. Hence the reliability expression is similar to parallel structure as in constraint (1.2.25). Constraint (1.2.26) computes the reliability of the \( j^{th} \) alternative for module \( i \). Constraint (1.2.27) ensures that only one
version will be chosen for any particular alternative, which can also be the dummy version. The last constraint ensures not all the selected alternatives for any module are dummies.

**Optimization Model 2**

A very common problem associated to the use of COTS component is that some of the alternatives available for one module may not be compatible with some alternatives of another module. This issue must be considered while formulating the optimum component selection problem. None of the models discussed so far accounts for it. The model 2 formulated due to kapur et al, (2003) accounts the compatibility of the module in forming the model. The additional constraint included in the optimization problem of model 2 are

**Problem (1.2.P10)**

\[ x_{gsq} - x_{huqc} \leq M \gamma_i \]

\[ q = 2, \ldots, V_{gs} ; c = 2, \ldots, V_{hu} ; s = 1, \ldots, m \]

\[ \sum_{t=1}^{d} \gamma_i = d - 1 \]

where

\[ d = (V_{gs} - 1)(V_{hu} - 1) \]

The constraint (1.2.28) and (1.2.29) make use of binary variable \( \gamma_i \) to choose one pair of alternatives from among different alternative pairs of modules. If more than one alternative compatible component are to be chosen for redundancy, constraint (1.2.29) can be relaxed as

\[ \sum_{t=1}^{d} \gamma_i \leq d - 1 \]

1.2.5 An Optimization Framework for “Build-or-Buy” Decisions in Software Architecture [Cortellessa et al, (2008)]
The authors introduced a framework that helps developers to decide whether to “build-or-buy” components for some software architecture. Each component can be either purchased, or probably adapted to the new software system, and/or it can be developed in-house. This decision theory is termed as “build-or-buy” concept that affects the software cost as well as the ability of the system to meet its other requirements.

**Assumptions**

1. The cost of an alternative is the development cost, if developed in house; otherwise it is the buying price for the COTS product. Reliability data set is given for COTS components are known.

2. The Cost and reliability of an in-house component can be specified by using basic parameters of the development process, e.g., a component cost may depend on a measure of developer skills, or the component reliability depends on the amount of testing.

3. Different COTS alternatives with respect to cost, reliability and delivery time of a module are available.

4. Different in-house alternatives with respect to unitary development cost, estimated development time, average time and testability of a module are available.

**Notations**

\[ C_{ij} \]
\( C_{ij} \) \( j^{th} \) instance of the \( i^{th} \) COTS component, \( j = 0 \rightarrow \) in-house developed instance

\[ C_{ij} \]
\( C_{ij} \) Cost of \( j^{th} \) instance of the \( i^{th} \) COTS component

\[ d_{ij} \]
\( d_{ij} \) Delivery time of \( j^{th} \) instance of the \( i^{th} \) COTS component

\[ t_i \]
\( t_i \) Estimated development time for in-house component

\[ \tau_{ij} \]
\( \tau_{ij} \) Average time required to perform test case is assumed for in-house component

\[ s_i \]
\( s_i \) Average number of invocations of \( i^{th} \) component (COTS/In-House)

\[ \mu_{ij} \]
\( \mu_{ij} \) The probability of failure of \( j^{th} \) instance of the \( i^{th} \) COTS component on demand (in one execution) [Trivedi, (2001)]
$C_{i0}$  
In-house developed component

$C_i$  
Unitary development cost for in-house component

$\pi_i$  
Probability that a single execution of software fails on a test case chosen from a certain input distribution. [Voas et al, (1995)] Testability for in-house component.

$N_i^{tot}$  
Total number of tests performed on the $i^{th}$ in-house component

$N_i^{suc}$  
Number of successful tests performed on the $i^{th}$ in-house subject to $N_i^{suc}=(1-\pi_i)N_i^{tot} \forall i=1,2,...,n$

$\rho_i$  
Reliability of $i^{th}$ in-house component

$y_i$  
Indicator is $\begin{cases} 1 & \text{if the } i^{th} \text{ component is in-house developed} \\ 0 & \text{otherwise} \end{cases}$

$x_{ij}$  
Indicator is $\begin{cases} 1 & \text{if the } j^{th} \text{ COTS instance of the } i^{th} \text{ component is chosen} \\ 0 & \text{otherwise} \end{cases}$

$f_i$  
Average number of failures of the $i^{th}$ component

**Delivery time** of the whole system has given a maximum threshold ($T$). In case of a COTS product the delivery time is simply given by $d_j$, whereas for an in-house developed instance the delivery time shall be expressed as $t_i+\tau_iN_i^{tot}$. Therefore, the following expression represents the delivery time $T_i$ of the component $i$

$$T_i = (t_i+\tau_iN_i^{tot})y_i + \sum_{j=1}^{m}d_jx_{ij}$$

It has been assumed that manpower is available to independently develop in-house component instances. Therefore, the delivery constraint can be reformulated as

$$\max_{i=1,...,n} (T_i) \leq T$$

which can be decomposed in the set of constraints $T_1 \leq T, ..., T_n \leq T$. 
Reliability of an in-house developed component is devised using Baye’s Theorem of probability as

\[ \rho_i = \frac{1 - \pi_i}{(1 - \pi_i) + \pi_i (1 - \pi_i)^{N_i}}; \quad 0 \leq \rho_i \leq 1, \forall i \]

Therefore, The in-house developed probability of failure on demand of an in-house developed instance can be expressed as \( 1 - \rho_i \)

Average number of failures \( f_i \) of the \( i^{th} \) component is given as

\[ f_i = (1 - \rho_i) \sigma_i y_i + \mu_i \sigma_i x_{ij} \]

By equation above, the probability that no failure occurs during execution of the \( i^{th} \) component is given by \( \Phi_i = e^{-f_i} \), which represents the probability of no failures occurrence in a Poisson distribution with parameter, \( f_i \). Therefore, the probability of failure free system is given by \( \prod_{i=1}^{n} \Phi_i \), so the reliability constraint with threshold \( R \) becomes \( \prod_{i=1}^{n} \Phi_i \geq R \)

The model formulation

Let \( S \) be a software architecture made of \( n \) components, with a maximum number of \( m \) COTS instances available for each component. \( T \) is assumed to be committed time to assemble system while ensuring minimum reliability \( R \) [Trivedi, (2001)] and spending a minimum amount of money. Therefore optimization model can be written as

Problem (1.2.P11)

Minimize \( COF = \sum_{i=1}^{n} \left( c_i (t_i + \tau_i N_{i}^{\text{tot}}) y_i + \sum_{j=1}^{m} c_{ij} x_{ij} \right) \) \hfill (1.2.30)

Subject to

\[ \max_{i=1, \ldots, n} (T_i) \leq T \] \hfill (1.2.31)
\[
\prod_{i=1}^{n} \Phi_i \geq R \quad \text{...(1.2.32)}
\]
\[
y_i + \sum_{j=1}^{m} x_{ij} = 1 \quad \text{...(1.2.33)}
\]
\[
x_{ij}, y_i \in \{0,1\}
\]

where equation (1.2.30) minimizes the cost objective function (COF), and equation (1.2.31) and (1.2.32) are delivery time and reliability constraints respectively. Equation (1.2.33) describing if an instance is bought (i.e. \( x_{ij} = 1 \)), then there is no in-house development (i.e. \( y_i = 0 \)) for each component \( i \). A model solution provides the optimal “build-or-buy” strategy for component selection, as well as the number of tests to be performed on each in-house developed component instances. The solution guarantees a system reliability over the threshold \( R \), a system delivery time \( T \) while minimizing the whole system cost.

1.2.6 A Hybrid Approach for Selecting Optimal COTS Products [Gupta et al, (2009)]

The authors in their work have formulated fuzzy multi-objective optimization model for selection of COTS components for development of a modular software system. The hierarchy structure of the software consists of three programs, four modules and eleven COTS products. Some specific functions of each program can call upon a series of modules, and several alternative COTS products are available for each module. Different weights are assigned to different modules using an AHP technique. The issue of compatibility amongst the COTS products is also discussed.

Notations

\( m \)  

The number of modules in the given software system

\( n_i \)  

The number of alternative COTS in the \( i^{th} \) module, \( i=1,2,\ldots,m \)

\( w_i \)  

the weight of the \( i^{th} \) module, \( i=1,2,\ldots,m \)

\( q_{ij} \)  

the quality level of \( j^{th} \) alternative COTS in the \( i^{th} \) module, \( i=1,2,\ldots,m; j=1,2,\ldots,n_i \)
Chapter 1: Introduction

The cost of the \( j^{th} \) alternative COTS in the \( i^{th} \) module, \( i = 1,2,\ldots,m; j = 1,2,\ldots,n_i \)

\[ C_{ij} \]

Indicator is:

\[ x_{ij} = \begin{cases} 1 & \text{if the } j^{th} \text{ alternative COTS of the } i^{th} \text{ module is chosen.} \\ 0 & \text{otherwise} \end{cases} \]

Therefore, multi-objective optimization model for selection of COTS component is formulated as

**Problem (1.2.P12)**

Maximize \( Q = \sum_{i=1}^{m} w_i \left( \sum_{j=1}^{n_i} q_{ij} x_{ij} \right) \) \quad \ldots(1.2.34)

Maximize \( C = \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} \) \quad \ldots(1.2.35)

subject to

\[ \sum_{j=1}^{n_i} x_{ij} = 1 ; i = 1,\ldots,m \] \quad \ldots(1.2.36)

\[ x_{ij} \in \{0,1\}, \forall i, j \]

\[ x_{rs} - x_{ut_k} \leq M y_k ; k = 1,\ldots,z \] \quad \ldots(1.2.37)

\[ \sum_{k=1}^{z} y_k = z - 1 \] \quad \ldots(1.2.38)

\[ y_k \in \{0,1\} ; k = 1,\ldots,z \]

Objective function (1.2.34) maximizes the overall quality of the software system, objective function (1.2.35) minimize the cost of the overall system. Equation (1.2.36) ensures only one COTS product is selected for one module. Equation (1.2.37) and (1.2.38) are the compatibility constraints for the COTS products.

In MOP model proposed above, the two objectives, i.e. the weighted quality and cost are considered to be vague and uncertain. The following non-linear S-shape membership functions are used to express the vague aspiration levels of the decision maker’s weighted quality and cost of the COTS software product.

- The membership function of the goal for weighted quality is given by
\[ \mu_Q(x) = \frac{1}{1 + \exp(-\alpha_Q(\sum_{i=1}^{m}w_i(\sum_{j=1}^{n_i}q_{ij}x_{ij}) - Q_m))} \]

where $Q_m$ is the mid-point (middle aspiration level for the weighted quality) at which the membership function value is 0.5 and $Q_c$ can be given by the decision maker based on his own degree of satisfaction for the weighted quality.

- The membership function of the goal for the cost is given by

\[ \mu_C(x) = \frac{1}{1 + \exp(-\alpha_C(\sum_{j=1}^{n_j}c_{ij}x_{ij}) - C_m))} \]

where $C_m$ is the mid-point (middle aspiration level for the cost) at which the membership function value is 0.5 and $\alpha_C$ can be given by the decision maker based on his own degree of satisfaction for the weighted quality.

Considering the Bellaman-Zadeh’s maximization principle [Bellman, (1970)] and using the above defined membership functions, the fuzzy multi-objective optimization model for selecting the COTS software products is formulated as follows

Maximize $\lambda$

Subject to

\[ \lambda \leq \mu_Q(x) \]
\[ \lambda \leq \mu_C(x) \]
\[ 0 \leq \lambda \leq 1 \]

And constraints (1.2.36) to (1.2.38)

1.3 STRUCTURE OF THESIS

The work presented in the thesis focuses on optimization models for component selection in design of a fault tolerant modular software system under consensus recovery block scheme. Various Crisp and Fuzzy Multi-objective Optimization models for selection of components are formulated and their solution methodology is presented with numerical examples.
Chapter 1: Introduction

The thesis is organized in chapters as follows:

Chapter 1: Introduction

Chapter 2: Optimal Component Selection for COTS Based Modular Software System

Chapter 3: Fuzzy Multi-Objective Approach to Component Selection for COTS Based Fault Tolerant Software System

Chapter 4: Optimal Component Selection for Fault Tolerance Modular Software System using Build-or-Buy Policy under Fuzzy Environment

Chapter 5: Fuzzy Multi-Objective Approach to Component Selection for COTS Based Modular Software System Incorporating Mandatory Redundancy for Critical Modules

Given below are the brief compositions of chapters:

Chapter 1 is an introductory chapter and gives a brief description of various concepts used in the thesis and literature survey. The topics included in section one are software engineering concepts, software development, life cycle process and methodology of component based software development. Fault tolerant techniques are also highlighted in this chapter. The second section reviews different optimization models in the literature for selection of components in design of a modular software system.

Chapter 2 Crisp optimization models for component selection of COTS based modular software system are developed in this chapter. This chapter focuses on development of COTS based software system. It is a system that has been built primarily by assembling a set of COTS software components. Respective developers of the components provide information about their quality in terms of reliability, cost and execution time. The parameters of reliability, cost and execution time are deterministic and crisp. Reliability is the major concern for development of such a system. Reliability of component based software system can be increased by adding
redundancy. A careful use of redundancy may allow the system to tolerate failures and thus improves system reliability. Such systems are called fault tolerant systems and are usually developed by integrating COTS components. Therefore, crisp optimization models for selection of COTS components are formulated for a fault tolerant modular software system under consensus recovery block scheme. In section one, multi-objective optimization model is formulated for maximizing reliability and minimizing cost subject to various constraints on component selection for a fault tolerant modular software system. In section two, we have investigated a problem of component selection for a fault tolerant modular software system incorporating execution time. The execution time taken by the software to perform a function is important to a developer as well as user. Long execution time for performing a function may cause dissatisfaction and lead to low productivity of the system. The objective of the optimization models developed in this section is to select COTS components in such a way so that the reliability of the software system is maximum and the deviational execution time is minimum under the constraint of component selection and budget. The issue of compatibility of modules is also discussed in both the sections as it is observed that some alternatives of a module may not be compatible with alternatives of other modules due to problems such as implementation, interfaces, and licensing. Numerical illustrations are provided to demonstrate the developed models.

Chapter 3 introduces fuzzy multi-objective approach to component selection for COTS based modular software system. The model formulation for the previous chapter requires an estimate of reliability and cost/execution time for various alternative COTS in the modules. Due to the changing environment, these estimates cannot be determined definitely because cost, execution time and reliability are affected by ambiguous and uncertain factors which cannot be measured precisely. Also the decision maker’s assessment about these estimates may be based on incomplete knowledge about the COTS product itself and other aspects (e.g. vendor’s credentials). Under such conditions; making a decision based upon crisp
model is not the best decision. Hence, we formulate fuzzy multi-objective optimization models for COTS software component selection based on imprecise aspiration levels, the decision maker may decide his aspiration levels on the basis of past experience and knowledge possessed by him. The problem is formulated for consensus recovery block fault tolerant scheme. In section one, we develop a fuzzy multi-objective optimization models for selecting COTS alternatives for modules with the dual objective of reliability maximization and cost minimization. Section two develops a fuzzy multi objective optimization model for selecting COTS components, based on maximization of system reliability and minimization of deviational execution time of the software under budgetary and other constraint on component selection. Issue of compatibility of modules is discussed in both the sections. Numerical illustrations are provided to demonstrate the models developed.

**Chapter 4** In this chapter fuzzy optimization models are developed for component selection for a modular software system incorporating build-or-buy strategy. In chapter two and three we focused only on COTS based modular software system. But at times there may be a situation when the components with the desired requirements are not available in the commercial market or there may be a situation that a COTS component is available, but it is economical to develop that component in-house. In both the cases the developer will go for in-house built component. While developing software, components can be both bought as commercial off-the-shelf (COTS) products, and probably adapted to work in the software system, or they can be developed in-house. This decision is known as build-or-buy decision. Since various alternatives are available to the software developer, he has to choose the right mix of components for the system he is developing. In section one, a fuzzy multi-objective optimization model is formulated for the selection of the alternatives (in-house or COTS) for modules to maximize the system reliability by simultaneously minimizing the cost under delivery time constraint. In section two of this chapter we have further broke down the approach for selection of components. If a COTS alternative is selected, then different versions are available for each alternative and only
one version will be selected for each alternative of a module. If a component is an in-house-built component, then the alternative of a module is selected. The model also determines the optimal redundancy level of the modules of the software system so as to maximize reliability and minimize cost under the delivery time and other constraints on component selection for fault tolerant systems. Issue of compatibility of modules is discussed in both the sections. Numerical illustrations are provided to demonstrate the models developed.

Chapter 5 presents fuzzy multi-objective optimization models for selection of COTS components with mandatory redundancy for critical modules. There are some modules in a system where failures can result in significant economic losses, physical damage or threats to human life. These modules are usually called critical modules. Therefore, there is a need for developing a system with a build in redundancy in the critical modules, so that if a particular component (alternative) in a module fails other one will take over and prevents the system from failure. In section one, a fuzzy multi-objective optimization model is developed for optimal component selection with the dual objective of reliability maximization and cost minimization of the overall system under the constraints on fault tolerance and criticality of modules. Section two aims at optimal selection of COTS components by minimizing the absolute deviational execution time and simultaneously maximizing the system reliability under the constraints on fault tolerance and criticality of modules using fuzzy approach. Finally in section three, mandatory redundancy for critical module has also been considered in designing fault tolerance system incorporating build-or-buy strategy under Consensus recovery block scheme.
Chapter 1: Introduction

Chapter-1. Introduction
- Background Study
- Thesis Contribution
- Literature Review

Chapter-2. Optimal Component Selection for COTS Based Modular Software System
- Crisp multi-objective optimization models are formulated for the selection of COTS components in design of a fault-tolerant modular software system

Chapter-3. Fuzzy Multi-objective Approach to Component Selection for COTS based Fault Tolerant Modular Software System
- Fuzzy multi-objective optimization models are formulated for the selection of COTS components in design of a fault-tolerant modular software system

Chapter-4. Optimal Component Selection for Fault Tolerance Modular software system using Build-or-Buy Policy under Fuzzy Environment
- Fuzzy multi-objective optimization models are formulated for component selection using build-or-buy strategy.

Chapter-5. Fuzzy Multi-Objective Approach to Component Selection for COTS based Modular Software System Incorporating Mandatory Redundancy for critical Modules
- Fuzzy Multi-objective optimization models are formulated for COTS products selection incorporating mandatory redundancy for critical modules to prevent system from a failure due to the modules that are critical in nature.

Figure 1.2.1: Structure and Relationship between the Chapters of Thesis
Conclusion of the work done, scope of future research and a reference list is given at the end of the thesis.

This thesis is based on the following research papers in the order of appearance in the thesis:


