Chapter 1

Introduction

With advent of the electronic digital computers, we have become dependent on the computers in our day-to-day life. Today computer hardware and software permeates our modern society. Computers are embedded in mobile phones, wrist watches, home appliances, automobiles and in almost all devices we use in our day to day life. The size and complexity of computer-intensive systems has grown dramatically during the past decade and the trend is certainly continuing. For example, in the telecommunications industry, operations for phone carriers are supported by hundreds of software systems with hundreds of millions of lines of source code. The demand for complex hardware/software systems has increased more rapidly than the ability to design, implement, test and maintain them. When the requirements for and dependencies on computers increase, the possibility of crises from computer failures also increases. The impact of these failures ranges from inconvenience e.g. malfunctioning of the device to economic damage, e.g. interruption of banking systems, to loss of life, e.g. failure of flight system or medical software. Needless to say, the reliability of computer systems has become a major concern for our society.

In the recent years, the cost of developing software and penalty cost of software failure have become the major expenses in a system. Failure of the software may result in an unintended system state of course of action. Locating software faults is extremely difficult and costly. A study conducted by Microsoft showed that it takes about 12 programming hours to locate and correct a software defect.

Software failures have led to serious consequences in business. A software problem contributed to a rail car fire in a major underground metro system in April of 2007 according to newspaper accounts. The software reportedly failed to perform as expected in detecting and preventing excess power usage in equipment on a new passenger rail car.
resulting in overheating and fire in the rail car, and evacuation and shutdown of part of the system. A review board concluded that the NASA Mars Polar Lander failed in December 1999 due to software problems that caused improper functioning of retro rockets utilized by the Lander as it entered the Martian atmosphere. Software design error and insufficient software testing caused an explosion that ended the maiden flight of the European Space Agency’s (ESA) Ariane 5 rocket, less than seconds after lift-off on 4th June 1996.[78]

With the advancement of technology, the cost of hardware is consistently decreasing. On the other hand, the cost of software is increasing. As a result, the ratio of Hardware and Software costs for a computer system has shown a reversal from the early years as shown in Figure 1.1 [38]

The main reason for the high cost of software is that software technology is still labor intensive. Software projects are often very large, involving many people and span of over many years. The advancements in the software technologies like Object Oriented
systems gave good opportunity to offer more facilities, flexibility and functionality to the user of the computers. This also increased the complexity of the Software development process. Today Software takes a dual role. It is both a product and a vehicle for delivering a product. As a product it delivers the computing potential embodied by computer hardware, or more broadly, by a network of computers that are accessible by local hardware. As a vehicle for delivering the products, software acts as the basis for the control of the computer, for e.g. Operating Systems, the communication of information, for e.g. Networks, and the creation and control of other programs, for e.g. Software tools and environments.

Today a huge software industry becomes dominant factor in the economies of the industrialized world. The lone programmer of an earlier era has been replaced by teams of software specialists each focusing on one part of the technology required to deliver a complex application. Over a period of time many lessons learned by the software developers and industry’s concern about software and the manner in which it is developed were addressed by adopting Software Engineering practice.

1.1. **Software Engineering**

Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation. Software engineering has to deal with a different set of problems than other engineering disciplines, since the nature of software is different. A software product is entirely a conceptual entity. Due to conceptual nature, there is an “intellectual distance” between the software and the problem the software is solving. This intellectual distance makes it harder for a person who understands the problem and to understand the software which is used to solve the problem. Due to the intellectual distance, a person responsible for maintaining the software has to spend considerable effort in just trying to understand the software that solves the problem. This makes the software maintenance very expensive. Because of this it is necessary to have a clearly defined process for developing the software.
Although there are some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different. Both these activities are people dependent but the relationship between people applied and work accomplished is entirely different. Hardware products exhibits very high failure rates early in the life cycle. Defects are then corrected and failure rates will drops to a steady state and after reaching its wear-out period where the hardware components start failing. The Figure 1.2 explains the hardware failure rate function for hardware.

![Failure curve for Hardware](image)

*Figure 1.2 – Failure curve for Hardware*

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 1.3. 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
1.3. Before the curve can return to the original steady state failure rate, another change is requested causing to spike again.

![Failure curves for Software](image)

*Figure 1.3 Failure curves for Software*

The basic goal of software engineering is to produce high quality software at low cost. The two basic driving factors are quality and cost. There are number of factors that determine software quality. McCall, Richards, and Walters [59] proposes a useful categorization of factors that affect software quality. These software quality’s are shown in Figure 1.4 below. These attributes focus on three important aspects of software product. [70]

- Operational characteristics (Product operations)
- Ability to undergo change (Product Transition)
- Adaptability to new environments (Product Revision)
Optimization Models for Component Selection in Designing of Modular Software System

Correctness: The extent to which a program satisfies its specification and fulfills the customer’s mission.

Reliability: The extent to which a program can be expected to perform its intended function with required precision.

Efficiency: The amount of computing resources and code required by a program to perform its functions.

Integrity: The extent to which access to software or data by unauthorized persons can be controlled.

Usability: The effort required to learn, operate, prepare input for, and interpret output of a program.

Maintainability: The effort required to locate and fix an error in a program.

Flexibility: The effort required to modify the program.

Testability: The effort required to test a program to ensure that it performs its intended function.

Portability: The effort required to transfer the program from one hardware and/or software environment to another.

Figure 1.4 McCall’s Product Quality Attributes

- **Reusability**: The extent of to which a program or its parts can be reused in other applications
- **Interoperability**: The effort required to couple one system to another

Of the above described attributes Reliability and Reusability are main focus areas for this thesis work.

### 1.2. Software Engineering Layers

IEEE defined software engineering as the application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of the software.

Any engineering approach must rest on an organizational commitment to quality [70]. Among the approaches to create quality culture, Total Quality moment, and Six Sigma implementation are very prominent. Whatever may the approach, the foundation is quality focus. The Figure 1.5 explains the layers of software engineering.

![Software Engineering Layers](image)

*Figure 1.5 Software engineering Layers*

At the bottom of the pyramid is quality focus. Unless there is no commitment and focus from every individual, who are involved in the development of a product, it is very difficult to deliver a quality product which is meeting the attributes mentioned in the previous sections. Therefore focus on the software is the fundamental necessity for any software development.
Process Layer is the glue that holds the technology layers together and enables rational and timely development of computer software. Without the process framework, it is difficult for the individual development groups and integrates them to achieve a single goal i.e. delivering software which meets the required functionality.

Software engineering methods encompass broad array of tasks that include communication, requirement analysis, design modeling, construction, testing and support. Standards like ISO 9000 help in defining these methods.

Tools will provide basic support in automated or semi automated support for the process and the method. For example Microsoft Office can be used for documenting the requirements and designs.

### 1.3. Software Development Life Cycle

In the 1960s and 1970s software development projects were characterized by massive cost overruns and schedule delays; the focus was more on planning and control. The emergence of Waterfall process to help tackle the growing complexity of development projects was a logical event [17]. A very simplified Waterfall model is described in the Figure 1.6 below.

![Waterfall Model](image)

**Figure 1.6 - Phases of Software development process**
System Requirements Phase: In this phase very high level functionality / specifications of the product is defined. This is done by the domain experts. This phase typically includes feasibility analysis where the feasibility of the project is analyzed and a business proposal is put forth with very general plan for the project and some cost estimates. The domain experts will perform this and asses what is feasible for this project.

Software requirement Analysis: Once the feasibility is established a detailed analysis is done in order to understand the problem which the software system is to solve. The functionality is explored in details and documented in this phase. All necessary system functional and non functional requirements are detailed in this phase. Systems Analysts will be performing this activity. Normally in this phase the functionality is divided into releases and scope for each release is defined.

Software Design: In this phase the requirements are further analyzed by the technical people. It has further subdivided in to small phases. System design, and detailed design. System design aims to identify the modules that should be in the system, the specifications of these modules, and how they interact with each other and to external applications to produce the required results. Once overall system design is ready individual project groups parallely start working on the high level design followed by low level design. During the detailed design phase, internal logic, algorithms and pseudo code is detailed. Broadly, system design defines what is to be achieved and detailed design defined how it is achieved.

Coding Implementation: In this phase the detailed designs are implemented using the language specified in the requirements. The goal of the coding phase is to translate the design in a given programming language. Both testing maintenance are affected by the coding phase. [38]. A well written code can reduce the testing and maintenance effort.

Software integration and formal system Test: Once the coding is done testing is performed. Testing is the major quality control measure employed during the software development. Its basic function is to detect errors. In next section testing is explained in detail.
**Operation and Maintenance:** During this phase, the system is installed and put to use after necessary training to the users. This phase continues until the system is operating in production in accordance with the defined user requirements. Any changes requested or defects found will be fixed during the maintenance stage.

In any product development design phase is an important phase where the system designed according to the specification. In the following section this phase is explained little more detail.

1.4. **Software Designing**

The design of a system is essentially a blueprint, or a plan for a solution for the system. In a project based or custom-specific development system, the software solution is designed to the specific needs of the customer. For the solution, the client will specify what he wants to achieve and also any specific need of the tools / products will also be specified in the requirements. The solution is finally developed using the tools specified by the client. Here the assumption is the customer has done the required analysis in selecting the required tools/products and gave the requirements. Hence, the solution is designed by using the tools/products/components.

Whereas in a product development, the process start with a general set of requirements based on the market requirements. Domain specialists, competitor’s products customers are the primary sources for defining the requirements. The developers analyze these requirements and build the product. They develop the products for global clients i.e. there are no specific clients for them. After developing the products it is offered as a solution for a specific purpose to market. Here the end users are more than one. Product development is never ending process as more and more new functionality is added at regular intervals to cater to the changing needs of the market and customization is possible.

New technological advancements like object oriented systems, made reuse and product integration very easy. It is possible to develop the components which can be reused in
any application where there is a need for such a functionality delivered by the component. These ready to use components are often called Commercial-Of-The-Shelf components (COTS). Apart from cost advantages, the time to market will be reduced by way of using the readily available components. Since these components are thoroughly tested the reliability of the new product will increase.

Because of the very evident advantages, the business organizations are moving towards more and more use of COTS components. The approach to the system development shifts significantly in COTS based system development when compared to traditional systems. The Software Engineering Institute has developed a process framework for working with COTS-based systems [5]. They have defining this new process driver as follows:

"CBS development is an act of composition. The realities of the COTS marketplace shape CBS development. CBS development occurs through simultaneous definition and trade-off of the COTS marketplace, system architecture, and system requirements." [65]. This fundamental change is shown in Figure-1.7

![Diagram: Traditional vs COTS based development approach](image)
1.5. **Software Testing**

Software testing is an important phase in SDLC. It plays a vital role in ensuring the quality of delivered product. A number of testing strategies have been proposed in the literature. [71]. Software is often referred as Verification and Validation (V&V). Verification refers to the set of activities that ensure that software correctly implements specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Verification and Validation methods are applied to various stages of software development life cycle. This was illustrated in Figure 1.8 below.

![Figure 1.8: Verification and Validation type for SDLC phases](image)

**1.5.1. Software Testing Strategy**

There are many testing strategies which are being used. The most widely used strategy is to test from the unit level to final end-to-end scenarios based on the stage at which the development phase is. Broadly the following are the types of testing adopted.
Unit Testing: Unit testing is performed generally by the developer. This is done at the time of coding phase. The main purpose of this testing is to test the piece of code called single unit to check that specific unit is working as per the design or not.

Module Testing: In this group units called a Module is tested collectively. This is also mostly done by the development group. Here the module which has set of units is tested to check whether the module as whole is working or not.

System Integration Testing: This type of testing is one step above the module testing. Integration of all modules in that system is tested in this phase. All integrations between the modules are tested thoroughly. This will be done by a separate testing group.

Functional System Testing: This is generally performed by the independent testing group. This is pure functional testing activity and here all specified functionality is checked along with the integration inter and intra application modules. In case of the new releases for the existing product, regression testing will also be conducted during the stage just to ensure the new functionality / changes does not effect the already existing and unchanged functionality.

Acceptance Testing: This is final testing phase of the SDLC before going for implementation. This is done by the users / owner of the software. In this phase the functionality is verified against the requirements and decided whether the delivered software is functional as per the requirement or not.
The Figure 1.9 will present the types of testing used for each development life cycle stage. This is called Verification and Validation Model or simply V-Model.

![V-Model Diagram]

**1.6. Software Reuse**

Use of Software has increased in every walk of life. It has intruded into the operation of the modern world. Internet, Banking, telecommunications, automobiles, space programs etc are few examples. The demand for more functionality led to the complexity in terms of the software and hardware. 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 re-used in the other applications with similar functional requirement. This led to an increase in the software development performance and thereby increases in the Business performance. In this case, success in improving business performance means that
products and services that rest upon a software base must get their software “Faster, Better and Cheaper”. [36].

- Faster, because the software has to meet dynamically changing the market requirement and to face the competition.
- Better, because the software has to serve the requirements of the process it is to support and later when serving its process, it has to do with few failures.
- Cheaper, because the software has to be less expensive to produce and maintain

Software reuse has led to development of components for addressing some standard functionality and thereby business can choose to use them directly to solve their business need. This will have certainly advantages as the cost of maintenance will come down. According to a study the following substantial gains are observed [36]

- Two to three time reduction in time to market
- Five to ten times reduction is defect density
- Five to 10 times reduction in Maintenance cost
- Around 15% to 75% reduction in overall software development cost

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” as the key to build applications rapidly. Examples of these technologies include Microsoft Visual Basic, ActiveX, Sun Java, and OMG’s Corba etc.

1.7. Commercial Off-the Shelf Software

Commercial off-the-shelf (COTS) software has been gaining popularity in the last few years. It is considered an economically viable method of integrating various software components to produce a new product. Many commercial and governmental organizations are now relying on COTS software rather than developing and maintaining their own programs. The use of COTS software has increased in the last decade. COTS
products seem to be less expensive than proprietary software to integrate into the infrastructure of an organization, whilst decreasing the development time for new products. Considering that large and complex systems require a massive amount of coding (for example about 300,000 lines of codes for cellular telephones) and that many programmers can only compose about ten lines of code per day, COTS products provide feasible alternatives to custom development efforts.

COTS software products have been defined as “commercial items that have been sold, leased, or licensed in a quantity of at least 10 copies in the commercial marketplace, at an advertised price. COTS software products include a description or definition of the functions the software performs, documented to good commercial standards, and a definition of the resources needed to run the software.”

Four issues need to be addressed when considering COTS software: “functionality and performance, interoperability, product evolution, and vendor behavior.” Functionality of COTS software determines the outcome of the system, while its performance depends on various factors including vendor's specifications. Many variables affect COTS programs interoperability with each other and with proprietary software, including the source of COTS software, presence or absences of standards, and the extent of use of open architecture. Vendors' commitment to upgrade COTS products and the software evolution cycle is usually independent of users' system requirements. Finally, each vendor's ability, willingness, resources, and attitude determine their behavior and conduct towards their customers and the level of support that they provide.

Developing a new system based on COTS products involves acquiring different components, customizing various software packages to suit local requirements, and gluing the pieces together. The advantages include predictable license costs, shorter development time, broad and immediate availability, comprehensive functionality and potentially frequent upgrades. The reality, however, may be quite different. The disadvantages of using COTS products may consist of initial cost of purchasing, limited functionality, almost complete dependency on the vendor, difficult integration...
procedures, potential unreliability, incompatibilities among vendors, and limited license agreements.

1.8. Component Based Development

Large Systems are developed using the components. Components are available in the market to cater to various needs of the system. For example, in a Banking application, payment gateway softwares are available as components. Normally these components cater to a certain business/technical functionality and will be maintained by the supplier of the component. This will help reducing the cost and development time. Component Based Development (CBD) is very much similar to conventional object oriented systems. The CBD process starts when a software team establishes requirement and design, the team examines the requirements and determines which part of the system is direct amenable to composition rather than construction. Then if any COTS components are available in the market and they are suitable and economical the decision will be made to use those components. In addition to the functionality of the COTS components and cost of the component the team verifies the following attributes:

❖ Performance
❖ Reliability
❖ Usability

In case there are more than one component which is serving the same functionality, selection of the component becomes a complex task. In this thesis, reliability optimization models for selection of COTS components are discussed.

Douglas and Laurence Brooks [University of York] have presented a framework for selection of the COTS components. Their framework is given in Figure 1.10
When using the CBD selection of COTS component from the available alternatives is a key activity which will impact the reliability of the main system. One needs to use these attributes carefully. Components can be either bought as COTS (commercial-off-the-shelf) products, or probably adapted to work in the new software system, or they can be developed in-house. This is a “build-or-buy” decision that affects the software cost as well as the ability of the system to meet its requirements. A recent empirical study on COTS-based software development shows that this decision is part of new activities integrating the traditional development process, and it is always based on the experience of project members.

1.9. Software Reliability

Software reliability engineering is centered on a very important software attribute: Reliability. Software reliability is defined as the probability of failure-free software operations for a specified period of time in a specified environment. Software reliability
is generally accepted as the key factor in software quality since it quantifies software failures [56].

Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However software reliability is different from hardware reliability in the sense that software does not wear out, burn out or deteriorate i.e reliability does not decrease with time. Moreover software generally enjoys reliability growth as long as the software is not modified.

For achieving reliable software four technical methods are applicable. They are:

- Fault prevention
- Fault removal
- Fault tolerance
- Fault forecasting

1.10. Software Reliability Modeling

Software reliability modeling is an important aspect of SRE that has received most attention [56]. There are two types of software reliability models: the deterministic and the probabilistic. The deterministic models are used to study the number of distinct operators and operands in a program as well as the number of errors and the number of machine instructions in the program. Two well known models are:

- Halstead’s Software metric
- McCabe’s Cyclomatic complexity metric.

**Halstead’s Software Metric:** This is used to estimate the number of errors in the program. This is probably the best known technique to measure the complexity in a software program and amount of difficulty involved in testing and debugging the software.
McCabe's Cyclomatic complexity metric: This metric is used to determine an upper bound on the number of tests in a program. This a complexity measure of digraph based on the control flow representation of a program. Cyclomatic Complexity is a software metric which provides a quantitative measure of the logical complexity of a program by counting the decision points.

1.11. Fault Tolerant Software

In the real world developing we will not be able to guarantee error-free software. The reason is that the two basic ways of showing that software is correct, proof of program correctness and exhaustive testing, may never be practical. A way to handling unknown and unpredictable software failures is through fault tolerance [56]. There are many techniques adopted for achieving the fault tolerance. Some of the major techniques are given below.

- Recovery Block
- N-Version Programming
- Consensus Recovery Block
- Acceptance Voting
- N-Self Checking Programming

Recovery Block: This is the one of the earliest fault-tolerant software that used the multi version software approach. The following Figure 1.11 explains this.
Optimization Models for Component Selection in Designing of Modular Software System

Module 1 Module n
Input
Acceptance test 1
A1 A2 An

Success

Acceptance test n

System Failure

'Correct' Output

Figure 1.11 – Recovery Block

N-Version Programming: In the parallel execution of N independently developed functionality equivalent versions with adjudication of their outputs by a voter is proposed.

Module 1 Module n
Input
Acceptance test 1
M1 M2 M3 Mn

Module Outputs

Voter

System Failure

'Correct' Output

Figure 1.12 N-version Programming
Consensus Recovery Block: This is a hybrid system. It combines N-Version Programming (NVP) and Recovery block [RB] in that order.

Acceptance Voting: The output of each module is tested and if the acceptance test passes then only it is passed to Voter. The voter sees only those outputs which are passed. This means the number of outputs processed by voter may not be same.
N-Self checking Programming: This is a variant of N-Version programming. The variation is N-modules are executed in pairs and the outputs from the modules are compared. The accepted output is the compared with the output from the next pair.

Figure 1.14 Acceptance test

Figure 1.15 N-Self Checking Programming
1.12. Structure of the Thesis

The work presented in this thesis focus on reviewing some optimization models in component selection in designing of modular software systems and extensions of these models are proposed.

Chapter – 1

This is an introduction explaining the software engineering concepts and touches upon software development life cycle and discusses in detail about software design, need for COTS based software development systems. Concepts of Software testing and fault tolerant software are discussed in this chapter. Very brief details about software reliability and reliability modeling are also discussed.

When building an application it is common to decide whether to build in house or buy from market. Also during the architecture stage, a thorough analysis of requirements will be made to identify the need for any components which are already available in the market. It may seem very simple to choose the components but it is a very complex technical process. The following figure broadly explains the process and how this thesis is connected to various phases of component selection.
Chapter - 2

Software application development saw many changes in the last two decades. New methodologies and techniques such as object oriented development; componentization of the system became necessary as complexity and size of the computer applications have increased tremendously. Most of the work has been done to measure software complexity and for the predicting quality of the application. Complex functionality gave way to
Optimization Models for Component Selection in Designing of Modular Software System

development of modular software system. Further, in the last decade, to achieve the same functionality choice of application components are available for reuse from different vendors in the market. Choosing the right components without suffering the final application’s quality is a challenge because of choices available. So there is a necessity to use good quality measures for selecting the components. Coupling and cohesion are two software quality metrics that captures at least eight out of thirteen quality characteristics. Coupling and cohesion are attributes that summarizes the degree of interdependence or connectivity among inter and intra dependencies respectively. When used in conjunction with measures of other attributes, coupling and cohesion can contribute to an assessment or prediction of software quality. In an object oriented system software complexity is determined based on interactions of Class attributes, method used etc., and level of interactions. At each level, measures are identified that account for the cohesion and coupling aspects of the system. Most of the object-oriented cohesion metrics proposed in the literature define static cohesion at class level. Measurement of object-level dynamic cohesion however gives better insight into the behavioral aspects of the system. There has been a growing interest in component based software system (CBSS) development both in academia and in industry. In CBSS development, it is common to identify software modules first. Once they are determined, we need to select appropriate software components for each software module. Hence a critical review of some literatures using Coupling and cohesion metrics to select component for COTS based modular software system has been discussed.

In the COTS based development system, acquiring a component either developed in-house or purchased from the market there is a cost associated with that. For any organization, budgets are not unlimited. Developing the software with cost optimization in mind is an important goal of any organization. When we develop the application with finite budgets, it is necessary to utilize the resources in an optimized way without compromising on quality and reliability. In order to achieve this we need to have optimization models which will minimize the cost and maximize the reliability and also ensures that the application runs without fail. With this in mind, a new constraint on cost is proposed in this section with an intention of minimizing the cost of selecting the

Vijaya Mohan Ghantasala
Sri Krishnadevaraya University
modules is proposed. The concept of fault tolerant software can be incorporated in future work.

Chapter - 3

Software Application development has become a very complex activity because of the functionality demands and many advances in the technology. These advances gave many facilities to users but for the developers who developed these applications often pass through very critical decision making challenges. In the modern times, time-to-market plays an important role. Delivering the application with the required functionality and right quality and reliability is major task. Added to this the organizations have to utilize the budgets in a very optimal way. In earlier chapters it was discussed that COTS based software development is one of the way to develop applications quickly and with in budgets. However, during the designing the software architecture, organizations face a challenge of choosing a decision to Build or Buy a particular component. This depends on the available budget, time and criticality of the component. In this chapter, optimization models to help in build-or-buy decisions are discussed. Also optimization framework that supports the decision whether to buy software component or to build it in-house was is reviewed.

The problem discussed in this chapter is about the Build-or-Buy strategy for component selection. The main driver for the decision is Cost, Delivery time and Reliability. In taking the decision to buy or build, cost is no doubt the primary driver however we cannot compromise on reliability and timely delivery of the software. This means this problem can be reformulated as Bi-Criteria optimization.

Further developer has to deliver in the specified time for the required quality software at minimum cost. Of course cost plays an important role in software development but developer cannot compromise with quality of the software. Hence in practice the budget and quality (reliability) of the software fixed prior to its development. The problem further modified and reformulated by adding two additional constraints reliability and available budget.
Chapter - 4

In the modern computer applications demand for high quality and reliability has resulted in growth of size and complexity, and the trend will certainly continue in future. Fault tolerance improves the system reliability which incurs huge cost to the developer so it is necessary to carry a tradeoff between cost and reliability. Fault tolerant software assures system reliability by using protective redundancy at the software level. The two basic techniques for obtaining fault tolerant software are Recovery Block Scheme and N-Version Programming. Advancement of technology has made the use of Commercial-off-the-Shelf (COTS) products in order to meet the challenges faced by organization because the resources are limited and the manager has to maneuver within a tight schedule. In this chapter, several optimization models on software reliability and cost of modular software systems incorporating Recovery Block Scheme are reviewed. Firstly, the optimization models for selection of modules are formulated considering information on reliability and cost.

The objective is to maximize the reliability of the software satisfying a budget limitation after this we addresses the problem of optimal selection of components for a modular software system that is built by assembling a set of Commercial-off-the-shelf (COTS) components and the issue of compatibility is accounted. Bi-criteria problem using goal programming approach is discussed by maximizing the reliability and minimizing the cost of the system with an upper bound on cost and lower bound on reliability.

When we develop the application with finite budgets, it is necessary to utilize the resources in an optimized way without compromising on quality and reliability. In order to achieve this we need to have optimization models which will minimize the cost and maximize the reliability and also ensures that the application runs without fail. Therefore there is a need for developing the system with a built in redundancy for the critical modules so that even a particular critical module fails, the other one will takeover.
We can consider to introduce redundancy for the critical modules and also at the same time choose the modules which meet the required reliability criteria and cheaper to acquire and use. Further to achieve this, the Bi-criteria model discussed in this chapter was extended by way of using a constraint on Criticality of modules so as to achieve the effective redundancy for all critical modules and at least one effective alternative for non critical modules.

Chapter - 5

In Chapter – 5, several optimization models incorporating Consensus Recovery (CRB) were discussed. Problem of optimal selection of components for a modular software system under Consensus Recovery Block scheme that is built by assembling a set of COTS components is reviewed. Basic information on component reliability, cost and allowing the tradeoff between two factors was mainly considered in these models.

In this chapter optimization model for allocation of COTS software with redundancy for critical modules under the consensus recovery block scheme is discussed. The Bi-criteria model discussed in this chapter was extended by way of using the constraint on Criticality of modules so as to achieve the effective redundancy for all critical modules and at least one effective alternative for non critical modules.