CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Software products have significantly changed aspects of work and daily life and its presence is everywhere, i.e. in the office, car, the Internet, etc. With almost all electrical equipment having some kind of software and this percentage increasing, Software helps to run and support manufacturing industry, education, entertainment, health and financial services, economic analysis, research, management activities, and many other domains. Software development has become one of the largest industries globally. Software Engineering emerged as a discipline in the late 1960s and attained increased importance in the past 40 years. Its objective is to integrate other engineering disciplines to the development of software products to improve their quality. It is inherently complicated, multidisciplinary and multi-dimensional thereby making it a unique discipline that requires specific solutions by combining knowledge from different disciplines.

Requirements Engineering (RE) is “The area of Software Engineering that focuses on the RE process which involves understanding customer needs and expectations (requirements elicitation), requirements analysis and specification, requirements prioritization, requirements derivation, partitioning and allocation, requirements tracing, requirements management, requirements verification, and requirements validation” (Young 2003). The terms Requirements and Engineering were first tied
together by Alford (Alford 1977) in the development of SREM (Software Requirements Engineering Method). RE was first applied to information systems, and therefore was more focused towards organizational and application issues. RE research efforts have endeavored to incorporate an engineering approach to what was traditionally known as systems analysis. Originally, RE activities were only related to requirements analysis and specification, the core of the first stage of the Software Development Life Cycle (SDLC). In the 1990s, it was widely accepted that RE was a key process in the software lifecycle and the scope of RE has extended far beyond than mere system analysis. RE is now an established discipline and includes a variety of skills, processes, methods, techniques and tools. In this section, an introduction to the concepts and issues in RE which are relevant to this thesis work is provided.

1.2 SOFTWARE ENGINEERING

Software Engineering evolved as a discipline due to the need to deliver quality software on time and on budget. Software Engineering seeks to ensure the above by setting out a systematic and disciplined approach to its development. Current Software Engineering ideas are based on the following:

- Understanding the development processes, project management and the ability to measure, monitor and control software development,
- Understanding the problem to be solved, design methods required to solve it and the platforms used to implement them and
- Understanding the range of tools and techniques required to support the processes and how they are to be used effectively to support problems.
Comprehensive definition of Software Engineering given by Berry (1998) which describes different facets: “Software Engineering is that form of engineering that applies a systematic, disciplined, quantifiable approach, the principles of computer science, design, engineering, management, mathematics, psychology, sociology and other disciplines as necessary and sometimes just plain invention, to creating, developing, operating and maintaining cost-effective, reliably correct, high quality solutions to software problems.”

1.2.1 Software Development Life Cycle (SDLC)

A software process model is an abstract representation of a process from a particular perspective. Elements defined within it include the roles, activities, their logical order, techniques used, transition criteria for progressing from one activity to the next, etc. The process model provides the Software Engineer with a means to understanding, reasoning about, planning, monitoring and controlling software development once it is carried out in practice (Eberlein 1997).

The major goals of software process modeling are to (Curtis 1998):

(1) Ease understanding and communication. To implement this goal, the process model shall contain enough formalized information in its representation, to facilitate training.

(2) Support process management and project control. This goal suggests that a process model is developed on a project basis so that the project can be effectively monitored, controlled and managed.

(3) Provide analysis for process performance. A software process development environment is required to develop a process model. Such an environment also helps analyze the
performance of the process, such as productivity of
developers in terms of function points, number of defects
identified, etc.

(4) Provide automated execution support. To implement this
goal, it is necessary to have automated process elements, co-
operative work support, and a compilation of metrics and
process integrity assurance during process modeling.

(5) Support process improvement. The process modeling which
implements this goal requires the reuse of well-defined
software processes, comparison of alternative processes and
process development support.

Over the last forty years, numerous Software Development Life
Cycle models have been developed. Some examples of these models are the
Waterfall model, Spiral model and Evolutionary Development model all of
which have the following fundamental stages:

- Software Specification Stage: In this stage, customer
  requirements are converted to Functional and Non-
  Functional Requirements and constraints of the software
  system and documented in the final requirements
  specification.

- Software Design and Implementation: Software is designed
  and implemented according to the requirement specification.

- Software Verification & Validation: The software must be
  verified for correctness and validated with the customers to
  meet their needs.

- Software Evolution: The software evolves to meet changing
  customer needs and improve the quality of the product.
1.3 INTRODUCTION TO REQUIREMENT ENGINEERING

Requirements Engineering (RE) is an area of Software Engineering that deals with the discovery and specification of the objectives for the system under development. This is an extremely important activity since the measure of success of software systems is the degree to which they can satisfy their requirements. Many of the existing Software Engineering methods have concentrated on the design of the system and devoted relatively little attention to RE. As well, most RE approaches assume that the initial formulation of the requirements was given. However, activities that lead to the formulation of the initial requirements for the system-to-be were mostly ignored.

Goal-Oriented Requirements Engineering (Dardenne et al 1993) has become prominent method in RE. Goal-Oriented RE approaches e.g., KAOS (Dardenne et al 1993) and Tropos (Jaelson Castro et al 2002) devote attention to understanding the environment of the system-to-be (the organizational context) and the rationale (the “why”) (Yu and Mylopoulos 1994) for the system. This is usually referred to as the early phase of RE. The goals of the stakeholders (individuals or organizations that influence the system and/or are influenced by it) are analyzed, refined and later assigned to the components that are part of the system and the agents that are in its environment. These approaches are requirements-driven as opposed to being driven by design or implementation factors. Their reliance on goals makes Goal-Oriented RE methods and Agent-Oriented Software Engineering a great match.

Agent-Oriented analysis is central to RE since the assignment of responsibilities for goals and constraints among components in the software-to-be and an agent in the environment is the main outcome of the RE process (Van Lamsweerde 2000). Therefore, it is natural to use a Goal-Oriented RE approach when developing multi-agent systems. The goals of the stakeholders
are refined and then assigned to the agents, thus turning into the objectives of agents in a multi agent system.

The foundation for any project is requirements, which define what stakeholders (customers, suppliers, developers) look for in a new system along with what the system should undertake to satisfy that need. They are usually expressed in natural language and the challenge is to ensure that the need or the problems is understood and unambiguously without resorting to specialist jargon or conventions. Once communicated and agreed, requirements drive the project activity. However, the many varied needs of stakeholders may come into conflict. These needs may not be clearly defined at the start, may be constrained by factors outside their control or may be influenced by other goals which themselves change with time. Without a relatively stable requirements base, a development project will flounder.

Requirements provide both the “navigation chart” and the means of steering towards the destination. Requirements provide the basis for planning the development of a system and acceptance of the system on its completion. They are essential when sensible and informed tradeoffs have to be made and are vital when changes are necessary during the development process. Even as the problem to be solved and potential solutions are defined, the risks of failing to provide a satisfactory solution must also be assessed. Few sponsors or stakeholders will support product or systems development without a convincing risk management strategy. Requirements enable the management of risks from the earliest possible point in development. Risks raised against requirements can be tracked, its impact assessed and effects of mitigation/fallback plans understood before substantial development costs have been incurred.
Requirements therefore form the basis for:

- Project planning
- Risk management
- Acceptance testing
- Tradeoff
- Change control

1.3.1 The Importance of Requirements Engineering

There is an increasing awareness of the importance of RE. Numerous researchers emphasize that the RE process is an essential contributor to the overall quality of the software product based on empirical investigations and industrial experiences (Hoare 1981, Brooks 1987, Emam 2000). Interestingly, the CHAOS report published by the Standish Group also strongly supports this argument. According to (Chaos 1995), good RE practices contribute more than 42% towards the overall success of a project, much more than other factors. Similarly, improper RE practices account for more than 43% of the reasons why projects are late or over budget. RE can be divided into discrete chronological steps:

- Requirements Elicitation,
- Requirements Analysis and Negotiation,
- Requirements Specification,
- System Modeling,
- Requirements Validation,
- Requirements Management.
1.3.2 Types of Requirements

Requirements are categorized in many ways, with a few common requirement categorizations relating to technical management are given below:

**Customer Requirements:** Factual statements and assumptions defining system expectations with regard to mission objectives, environment, constraints, and efficiency measures and suitability.

**Architectural Requirements:** This explains what should be done through identification of required system architecture which includes structure and behaviour in a system.

**Structural Requirements:** This explains what must be done for identification of necessary system structure.

**Behavioral Requirements:** This elucidates action based on system behaviour identification.

**Functional Requirements (FR):** FR describes what should be undertaken through identification of necessary task/action/activity to be achieved. FR analysis is used as a top level function.

**Non-Functional Requirements (NFR):** NFR specifies criteria to judge system operation rather than specific behaviors.

**Performance Requirements:** The level to which a function must be implemented; usually measured on quality, quantity, timeliness/readiness parameters. Performance requirements are interactively developed across all identified functions during requirements analysis.
**Design Requirements:** Processes expressed in technical data packages regarding the “build to,” “code to,” and “buy to” requirements for products and “how to execute” are referred to as design requirements.

**Derived Requirements:** Implied or transformed requirements from a higher-requirement level are known as derived requirement.

**Allocated Requirements:** In this a requirement is established by dividing/allocating a high requirement level into multiple lower-level requirements.

**Software Requirement Specification:** It is a description of system behaviour yet to be developed and includes set of use cases (also known as called FR) which describe proposed user software. Additionally, SRS includes NFR which imposes constraints on design/implementation (design constraints, performance engineering requirements, quality standards)

1.3.3 **Requirements and Quality**

The consequences of having no requirements are many and varied. There is ample evidence of systems that failed because requirements were not properly organized. However, a system appears to work well at first, but if it is not the system that user want or need then it will be useless. It is interesting to consider the relationship between requirements and quality. The term “quality” may be understood in a variety of ways. Quality is “fitness for purpose” or adheres to requirements –providing satisfaction to the customer and thereby ensures stakeholders needs are considered.
RE acts as a complement to other management considerations, like cost and schedule; providing a focus on the delivery of quality. Every management decision is a compromise between cost, schedule and quality, the three inter-related axes. Since RE is a discipline that is incorporated from the start of the development lifecycle, the leverage on quality that can be exercised by proper requirements management is proportionately greater. Relatively little effort expended in early stages of development can reap dividends later. Improving requirements means improving the quality of the product.

1.3.4 Requirements and Modeling

The relationship between requirements management and system modeling should be understood. They are mutually supportive activities that should not be equated. For example, in a sandwich requirements management is the “bread and butter” of the development cycle. The “filling” provided by system modeling explains and exposes the analysis and design that led to subsequent layers of requirements. Requirement Modeling is a misnomer. It is the Model which is designed and not the requirements. Modeling supports the design activity and is where most creative work takes place. It assists the Engineer to understanding the system to decompose the requirements at a particular level into the next lower level. The requirements themselves are a complete snapshot of what is required at each level in increasing levels of detail. A particular model never says everything about a system – if it did, it would not be a model. For this reason, several different, possibly inter-related systems are used to cover a variety of different aspects. It is left to the expression of requirements – usually in textual form – to cover those aspects not modeled.

A model is a system abstraction that focuses on some aspects of a system to the exclusion of others. Abstraction is, in this sense, avoidance of
distraction – ignoring those details that, although important, are not relevant to a particular model. The advantage of this is that smaller amounts of related information can be collected, processed, organized and analyzed, applying various specific techniques pertinent to the aspects under study. Where a large amount of complex information has to be managed, modeling provides a means of zooming in, collecting together subsets of the data for a particular purpose and zooming out once more to appreciate the whole. It aids in maintaining a system-wide grasp through focusing on small amounts of information at a time. Models assist the requirements engineer in analyzing the requirements at a particular level so as to:

- Communicate with the customer and improve mutual understanding of the system to be developed
- Analyze the system to ascertain the presence of desired emergent properties (and the absence of undesirable ones)
- Determine satisfaction of requirements by deriving new requirements at a lower layer

1.3.5 Requirements and Testing

Testing is any activity that allows system defects to be detected/prevented, where a defect is a departure from requirements. Testing activities include reviews, inspections, analysis through modeling in addition to classical tests of components, subsystem and systems that are carried out. Because of the diversity of testing activities, the term qualification is used. Qualification should begin as early as possible, as waiting till the system is almost complete before testing can lead to expensive design changes and rebuilds. The earliest kinds of qualification action take place during the system design and include requirements reviews, design inspections and various forms of analysis carried out on models.
Figure 1.1 portrays the qualification strategy along a time-line below the V-model. Early qualification actions relate to the left side of the V-model and later ones to the test stages on the right.

![Figure 1.1 Qualification strategy and the V-model](image)

A single stakeholder requirement will give rise to a multitude of qualification activities at various stages of development. Where a requirement is satisfied through useful emergent properties, qualification of components alone is insufficient; tests have to be carried out at the level where emergent properties manifest.
1.3.6 Requirements in the Problem and Solution Domains

Systems Engineering is concerned with developing and managing effective solutions to problems. It is a staged process, vital for businesses in enabling them to produce the right product within acceptable time-scales and costs.

Table 1.1 Problem and solution spaces

<table>
<thead>
<tr>
<th>Requirement Layer</th>
<th>Domain</th>
<th>View</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholder</td>
<td>Problem domain</td>
<td>Stakeholder’s view</td>
<td>State what the stakeholders want to achieve through use of the system. Avoid reference to any particular solution.</td>
</tr>
<tr>
<td>System requirement</td>
<td>Solution domain</td>
<td>Analyst’s view</td>
<td>State abstractly how the system will meet the stakeholder requirements. Avoid reference to any particular design.</td>
</tr>
<tr>
<td>Architectural</td>
<td>Solution domain</td>
<td>Designer’s view</td>
<td>State how the specific design will meet the system requirements.</td>
</tr>
</tbody>
</table>

Early in the process, the definition of the requirements for the product to be built is of paramount importance. From a management and engineering point of view, a clear distinction should be made between “problem domain” and “solution domain”. Stages of development associated with the highest levels of system description – statement of need, usage modeling and stakeholder requirements – should be rooted in the problem
domain, whereas subsequent layers, starting with system requirements, operate in the solution domain. Table 1.1 portrays the ideal boundary between the problem and solution domains and the roles that top requirements layers play. There is an important principle of abstraction at play here. The initial statement of capability should state no more than is necessary to define the problem and avoid any reference to particular solutions. This allows freedom to the systems engineers to devise the best solution without preconceived ideas.

Modeling assists in the derivation of the next layer of requirements and tends to consider possible solutions, even at a high level. To avoid inappropriate solution bias, rather than focus on the system in question, early modeling should focus on the immediately enclosing system. For instance, if a radio system is being developed for a sailing boat, then early modeling should focus on the vessel and not so much on the radio. This leads to a statement of the problem to be solved in the context of the enclosing solution.

Stakeholders frequently express the problem in terms of a preconceived solution. It then becomes the Requirements Engineers’ job to determine whether there is a good reason for mandating a particular solution or whether it is an unnecessary constraint. For example, the customer starts by trying to procure traffic lights; the supplier asks questions that lead to an understanding of the underlying objectives – maximize traffic flow and minimize risk for drivers and pedestrians – leading to a solution-independent expression of the problem; the reasons for the choice of solution are now better understood and perhaps confirmed through appropriate modeling, leading to a precise and well-informed specification of the abstract solution. When it comes to procuring systems, a judgment needs to be made as to whether to procure against the problem domain (stakeholder requirements) or against the abstract solution domain (system requirements). Often the nature
of the solution is known in advance and it makes sense to procure against system requirements framed within the solution. However, even if procuring against a particular solution, the discipline of capturing a statement of the pure problem prior to a solution still offers important advantages.

Without a clear distinction between problem and solution, the following may result:

- Lack of understanding of the real problem
- Inability to scope the system and understand which functions to include
- Domination of debate about the system by the developers and suppliers, because the only descriptions of the system are expressed in terms of solutions
- Inability to find optimal solutions due to lack of design freedom.

For these reasons, it is necessary to make distinctions between stakeholder and system requirements, in terms of how requirements are to be captured, modeled and expressed.

1.3.7 Requirement Traceability

In RE context, traceability understands how high-level requirements – objectives, goals, aims, aspirations, expectations, needs – are transformed into low-level requirements. It is primarily concerned with the relationships between layers of information.
In a business context, one may be interested in how

- Business vision is interpreted as
- Business objectives are implemented as
- Business organization and processes.

In an Engineering context, the interest may focus on how

- Stakeholder requirements are met by
- System requirements are partitioned into
- Subsystems are implemented as
- Components.

Using traceability can contribute the following benefits:

- Greater confidence in meeting objectives. Establishing and formalizing traceability engenders greater reflection on how objectives are satisfied.
- Ability to assess the impact of change. Various forms of impact analysis become possible in the presence of traceability information.
- Improved accountability of subordinate organizations. Greater clarity of how suppliers contribute to the whole.
- Ability to track progress. It is notoriously difficult to measure progress when all that is done is creating and revising documents. Processes surrounding traceability allow precise measures of progress in the early stages.
• Ability to balance cost against benefit. Relating product components to requirements allows benefit to be assessed against cost.

Traceability relationships are linked in a free for all way; one lower level requirement may be linked to several higher level requirements and vice versa. The simplest way to implement traceability is to link requirements statements in one layer with statements in another. Requirements management tools typically allow such linking through drag-and-drop between paragraphs of documents. The links are like hyperlinks in web pages, but should be traversable in either direction. The direction of the arrows follows a particular convention: information traces back to the information it responds to. There are a number of reasons for this convention:

• It usually corresponds to the chronological order in which information is created: always link back to the older information.

• It usually corresponds to access rights due to ownership: one owns the outgoing links from a document, someone else owns the incoming links.

Various forms of traceability analysis can be used to support Requirements Engineering Processes, presented in Table 1.2.
## Table 1.2 Types of traceability analysis

<table>
<thead>
<tr>
<th>Type of Analysis</th>
<th>Description</th>
<th>Processes Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impact Analysis</td>
<td>Following incoming links, in answer to the question: “What if this was to change?”</td>
<td>Change Management</td>
</tr>
<tr>
<td>Derivation Analysis</td>
<td>Following outgoing links, in answer to the question: “Why is this here?”</td>
<td>Cost–Benefit Analysis</td>
</tr>
<tr>
<td>Coverage Analysis</td>
<td>Counting statements that have links, in answer to the question: “Have I covered everything?” Most often used as a measure of progress</td>
<td>General Engineering Management Reporting</td>
</tr>
</tbody>
</table>

Impact analysis is used to determine which artifact in development might be affected when a selected artifact changes. The impact is potential; creative analysis has to be carried out to determine the exact nature of the impact, if any. Derivation analysis works in the opposite direction to impact analysis. A low level artifact – such as a requirement, design element or test – is selected and the traceability links are used to determine what higher level requirements have given rise to it. Elements in the design that do not trace back are potentially adding cost without benefit. Finally, coverage analysis can be used to determine that all requirements do trace downwards to lower layers and across to tests. The absence of such a trace is a fairly certain indication that the requirement will not be met or tested. The presence of a link does not ensure that the requirement will be met – that again requires creative engineering judgment.

Coverage can also be used to measure progress: how far have systems engineers got in responding to stakeholder requirements? When the
task of writing systems requirements in response to stakeholder requirements is given to engineers they write system requirements and link them back to the stakeholder requirements to which they are responding. At any stage of the task, engineers’ progress can be measured in terms of the percentage of stakeholder requirements that have been met. This is useful tool during the early stages of development. The same principle can be used to measure progress in planning tests.

1.4 TRACING DEFECTS TO REQUIREMENTS

Detecting defects in software product development requires serious effort, so it is important to use efficient and effective methods. The term ‘defect’ relates to one or more primary faults in an artifact like code.

As software unreliability has become a major bottleneck in improving system reliability, developers feel it is essential to detect and eliminate software faults at the earliest. The software industry uses methods such as testing, voting (also known as back-to-back testing), inspections, walk-throughs, self-checking code, and data-flow analysis. While it is suggested that some or all of the above techniques should be used to develop the highest quality software, software development projects will always have to be carried out with limited resources. In order to maximize software development productivity, practitioners should be guided by empirical and objective data on the cost-effectiveness of the proposed methods. Unfortunately, most of the empirical studies to date were conducted under different conditions and examined programs of different size and application domains.

Defect tracking locates defects in a product (through inspection, testing, or from customer feedback) in engineering, making new product versions which remove defects. Defect tracking is important in Software
Engineering as complex software systems have many defects and it is challenging to manage, evaluate and prioritize them. Defect tracking systems need a computer database to store and manage defects.

Defect prevention aims to identify defects and correct it so that it is not repeated. Defect prevention is implemented through preparation of an action plan to minimize/eliminate defects, generate defect metrics, define corrective action and analyze defect’s cause.

Defect prevention is achieved by the following steps:

- Calculating defect data through reviews using execution phase test logs; this data should segregate/classify defects based on root causes to generate defect metrics highlighting problem areas

- Identify improvement strategies.

- Escalate issues to senior management/customer where necessary.

- Drawing up an action plan addressing major defects; improving development process which is to be reviewed regularly for effectiveness and modified if ineffective.

- Undertake periodic peer reviews to verify that action plans are adhered to.

- Producing age based regular reports. If the defect age for a specific defect causes concern, focused action is required to resolve it.

- Classifying defects into categories like critical, functional, and cosmetic defects.
1.5 FUNCTIONAL REQUIREMENTS AND NON FUNCTIONAL REQUIREMENTS

Delivering the code on time and on budget is a main concern for Software developers. With the focus on delivering on time, NFR such as reliability, security, maintainability, portability, accuracy, are compromised. With the growing complexity of the software and higher quality software required, Non-Functional properties cannot be considered as secondary issue. Due to non-compliance of NFRs, many systems fail or fall into disuse. These Non-Functional aspects have been treated as properties or attributes. Earlier works viewed them as properties or attributes of the finished software product which were evaluated and measured only on completion. NFRs have always been a concern among Software Engineering researchers and now it is dealt with from the earliest stages of the software development process (Chung et al 2000) and then throughout the entire life cycle.

NFRs are also known as Quality Requirements (Boehm Barry and In Hoh 1996) and unlike FR, NFRs state constraints to the system and of qualities of a system like accuracy, usability, safety, performance, reliability, security. Hence, it can be stated that while FRs state “what” the system must do, NFRs constrains “how” the system must achieve the “what”. Due to this a NFR is always linked to a FR (Chung et al 2000).

FR addresses specific problems and is therefore usually implemented through specific localized modules or components. It is formalized when necessary but is usually stated informally. Conversely, NFRs define global constraints on a software system, on a FR, on the development process. As it arises from all parts of the system and from their interactions, it is considered to be global. There are well developed notations for specifying FRs, e.g., Structured English, Use Cases/UML, and various
formal methods approaches. In contrast, NFRs are more elusive to specify or to illustrate formally. Thus, they are normally stated informally in requirements documents. This leads to difficulty while enforcing NFRs during development and problematic to be assessed by the customer earlier to delivery. NFRs are interpreted differently under different contexts leading to difficulties in evaluating it by the stakeholders.

FR and NFR frequently appear together, as the Non-Functional needs to refer to the Functional. Suppose a system is dealing with the control an Automatic Teller Machine (ATM). There would be a Functional Requirement “The system must allow the customer to withdraw money”. One or more NFRs associated to this FR, such as “For security reasons, transactions should be completed within 5 minutes requiring a response time of less than 3 sec in at least 90% of the cases” is defined. But the system must also be secure when transmitting data, thus encryption should be used which may compromise the 3 sec goal (Cysneiros et al 2004).

A common method to understand the way a NFR constrains the FR is by decomposing the NFR into sub-goals. These sub-goals are represented by a graph structure based on And/Or trees which are used in problem solving methods. The procedure continues as long as the NFR is considered to be sufficiently satisfied by the requirements engineer. Another important characteristic of NFRs is that different NFRs may lead to contradictory solutions which are to be dealt with.
1.5.1 Why Non Functional Requirements?

There has been work showing that complex systems must deal with Non-Functional aspects (Dardenne et al 1993, Mylopoulos et al 1992, Chung et al 1995). These Non-Functional aspects should be dealt within the process of NFR definition. Errors due to omission of NFRs or improper dealing with them are among the most expensive type and most difficult to correct (Mylopoulos et al 1992, Ebert 1997, Cysneiros et al 1999). Recent work points out that early-phase RE should address organizational and NFRs, while later-phase focus on completeness, consistency and automated verification of requirements.

A more serious problem related to NFRs was seen in the London Ambulance Service Report (Finkelstein et al 1996, Breitman et al 1999). The London Ambulance System was deactivated just after its deployment because, among other reasons, NFRs neglected during system development include reliability (vehicles location), cost (best price emphasis), usability (poor information control on screen), and performance (system functioned but performance was unacceptable).

1.5.2 Approaches for dealing with Non Functional Requirements

Most prior work on NFRs measured how much a software system was according to a NFRs set, through some quantitative analysis (Boehm 1978, Fenton et al 1997, Keller et al 1990), which provided predefined metrics to assess how much a software object met a specific NFR. Works are proposed for techniques to justify requirements design decisions on requirements inclusion/exclusion which impact software design. Unlike metrics approaches, the latter approach is concerned with making NFRs a relevant and important in software development.
Boehm suggested a knowledge base where NFRs are accorded priority through stakeholders’ perspectives dealing with NFRs at high abstract levels. Kirner et al describe properties for six NFRs from real-time system domain: performance, reliability, safety, security, maintainability and usability. This provides heuristics on application of identified properties to meet NFRs and to measure NFRs later. But it lacks a larger approach applicable to other NFRs, in real-time or other domains. A major advance was when NFRs were treated as competing goals extensively refined and traded off among others to arrive at worthy solutions.

NFR Framework is one work which deals with NFR beginning with early software development stages through a broader perspective. The Framework (Chung et al 2000) views NFRs as goals which might come into conflict with each other and should be represented as soft goals to be satisfied. The soft goal concept was thought up to cope with NFRs abstract and informal nature. Each soft goal is decomposed into sub-goals represented through graph structure inspired by And/Or trees used to solve problems. This continues till the requirements engineer considers a soft goal is satisfied. Initially vague, NFRs were later operationalized with regard to implementable techniques. Operationalizations can be thought to be FRs that have arisen due to a need to meet NFRs. But important as it is to get a well-formed and as-complete-as-possible requirements set, it should understand and systematize how requirements drive the rest of software development, especially during a design phase.

NFR approaches can be classified into product-oriented and process-oriented. The former concerns measuring how much software complies with Non-Functional Requirements. They do not prevent problems but help to evaluate how much it complies with Non-Functional needs. The latter approach focuses on software development process, aiming to help software engineers looking for alternatives to meet NFRs when developing
software. It also justifies design decisions. The process-oriented approach follows guidelines like ISO 9126 or uses a Goal-Oriented approach like KAOS framework (Chung et al 2000, Van Lamswerde 2001). An advantage of a Goal-Oriented approach is that it can model/reason about both FR and NFRs.

1.5.3 Dealing with Non Functional Requirements

Handling NFRs includes varied activities like eliciting, modeling, and analyzing with each having its own challenge. Eliciting NFRs requires domain understanding and gathering organizational knowledge. As NFRs are generally stated are unclear in stakeholders’ minds as FRs, eliciting them is challenging. Existing NFRs knowledge can be used whenever possible for NFRs elicitation. Once elicited, they should be modeled and modeling them ensures that they are organized for better visualization and understanding. It helps Software Engineers analyze NFRs as this involves reasoning how well each NFR is satisfied and to iron out possible conflicts. This could mean that NFRs need further refining and also bring to light new conflicts. Alternatives to deal with conflicts allowing tradeoffs among stakeholders should be found out. Each alternative must be evaluated to express to which degree it can introduce positive/negative influence on one or more NFRs. For example, satisfying security could require encrypting mechanism, where its use could be in conflict with performance needs.
1.6 MOTIVATION

In general, it is assumed that the requirements for a system are clearly specified before its design and implementation can start. Mistakes during requirement process cause undue cost overruns, delays and many project failures. Requirements can be broadly classified into FR and NFR. FR deals with requirements that affect the functionality of the system whereas NFR deals with requirements that constrain the system. NFRs are also part of FR and may appear regularly when FR is being elicited. Since NFR are part of FR, its discovery is sometimes missed out during the initial stages of the SDLC. The different views of stakeholders on a NFR will also avoid clarity on the system wide NFR. Tracing NFR from FR is a time consuming task and requires experienced analyst to identify NFR hidden in business requirements.

Techniques using data mining have been proposed for retrieval of NFR from FR with fairly good results. However, effort has not been made in the direction of improving Information Retrieval (IR) algorithms nor has work been carried out in the areas of identifying NFR from FR based requirement elicitation.

1.7 OBJECTIVES OF THE THESIS

NFR of software evaluates the operation of the system and imposes constraints on design and implementation to maintain the overall quality. Incorporating RE to identify NFR in early stages avoids ambiguities, conflicting requirement and other defects.

Number of automated tools is available for identifying NFRs. The efficacy of the RE methodology is its ability to capture NFR in an efficient manner. This research can be broadly classified into the following sections.
• Investigation of classification accuracy of Meta learning algorithms including Logitboost and Bagging.

• Investigate effectiveness of Neural Network for identifying NFR features.

• Propose an improved Neural Network architecture Hidden Layer Genetic Optimized Recurrent Neural Network (HLGO-RNN) and Hidden Layer Genetic Optimized General Feed Forward Neural Network (HLGO-GFFNN) for identifying NFR features.

• Propose a feature extraction technique based on NFR Repository

• Propose a Hybrid Genetic Algorithm Continuous Ant Colony Optimization (HGACACO) which incorporates the features of Genetic Algorithm and Ant Colony Optimization.

1.8 THESIS OVERVIEW

This thesis is organized into six chapters. The first chapter dealt with the introduction to the subject of RE; the requirement process, defect tracing in requirements, the problems associated with it, background of the problem, and objectives of the study and contributions of this thesis. Second chapter deals with literature survey on RE, literature review on defect tracing to requirements, information retrieval for RE and review on NFR.

In the third chapter, the performance of Meta learning algorithms for classifying NFR associated with the requirement document is studied. Chapter four deals with the performance of Generalized Feed Forward Neural
Network and Recurrent Neural Network for classification of NFRs. Chapter four also proposes a genetic algorithm technique to optimize the learning rate and the momentum.

Chapter five details a feature extraction technique for NFR classification. To optimize the learning rate and the momentum a Hybrid Genetic Algorithm Continuous Ant Colony Optimization (HGACACO) technique is proposed. The proposed method is benchmarked with the dataset available in the promise data repository. Chapter 5 also introduces a new dataset obtained from M.E students over a period of five years and the proposed techniques evaluated. Chapter six concludes this thesis with directions for future work.