Functional Requirements and Non-Functional Requirements: A Survey

Jun Ying Zhou

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada
April 2004

Your file Votre référence
ISBN: 0-612-91163-2
Our file Notre référence
ISBN: 0-612-91163-2

Abstract

Functional Requirements and Non-functional Requirement: A Survey

Jun Ying Zhou

Software impacts almost every aspects of modern society. Software development process is a coherent set of activities for software modeling and associated artifacts. Requirements are basis of software systems. As the first phase of the software development process, requirements set reasonable targets to be achieved.

The purpose of this report is to study the role of requirements in software development process, and to survey the methods for specifying functional and non-functional requirements (NFRs). In this report the concepts of functional and non-functional requirements are introduced. The problems of functional and non-functional requirements lay on one of the following aspects: *identify*, *document*, *validate* and *verify*, so this report presents some answers to these questions through discussion and comparison of many process and methods. We also introduced some NFR frameworks are introduced and overviewed some tools used to support requirements.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

In this major report, we have stressed the importance of the role of requirements in software development process, and have contributed to the classification and standardization of the software non-functional requirements.

1.1 The Importance of Requirements in Software Engineering

Software impacts almost every aspects of modern society. Software development process is a coherent set of activities for software modeling and associated artifacts. Software Engineering is a discipline for the systematic construction and support of software products so they can safely fill the uses to which they may be subjected. Requirements are basis of software systems. As the first phase of the software development process, requirements set reasonable targets to be achieved.

Mistakes introduced at this phase of software development lead to failures and costly maintenance phase of the product. The Standish Group's CHAOS Reports from 1994 and 1997 [STA97] established that the most significant contributors to project failure relate to requirements. Another study [COM97] of 500 IT managers in the U.S. and U.K. reported that 76 percent of the respondents had experienced complete project failure during their careers, and most frequently named cause of project failure was "changing user requirements."

In software industrial practice, the high cost of development process of large-scale software has put emphasis on the need to prevent problem occurrence, rather than fix errors during the later phases of a software life cycle. As the above says, if we want to build a "right" software system, that is, if we want to build a software system just as what customers expect, we must start with "correct" requirements. The major objective of the requirements engineering is defining the purpose of a proposed system and outlining its external behavior. Requirements generally

express what an application is meant to do. They do not attempt to express how to accomplish these functions. The set of requirements for the system should describe the functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge. The functional requirements for a system describe the functionality or services that the system is expected to provide. Non-functional requirements, as the name suggests, are those requirements that are not concerned with the specific functions delivered by the system. They may relate to emergent system properties such as reliability, response time.

Now many methods in collecting, tracing, reflecting requirements are produced, which we could use to trace, control, manage requirements, but there are still work to do. First, non-functional requirements were not paid much attention to until recently, that is to say, whether in research or in practice, non-functional requirements were not treated seriously even in very recent. Second, in order to build software quality into software system, we must develop new software development process which should have much difference from exist ones because we must make non-functional requirements as a natural part of the process.

1.2 Purpose and Problem Statement

Many mature methods on dealing with functional requirements have been used for a long time and proved to be effective. Functional requirements are often described as "should do..." or "should be...", that is, they are easier to validate and verify. But to the non-functional requirements, it is hard to define an accurate target. For example, customer may require a "Friendly" software system. We recognize that in fact the user requires the "Usability" of the software system interface, perhaps also the "Effiencency", but it's hard to define an accurate target to achive. Concequently, valiadation and verification of non-functional requirements are more difficult than of functional requirements. Therefore, methods must be used to make this requirement to be visual and clear.

In this report, we focus on the problem stated above. The purpose of this report is to study the role of requirements in software development process, and to survey the methods for

specifying functional and non-functional requirements (NFR). Some NFR frameworks are introduced and tools used to support requirements are overiviewed. "How to identify requirements", "how to document requirements" and "how to deal with requirements" are always the most common questions in front of us, so this report presents some answer to these questions through discussion and comparison of many process and methods.

1.3 Report Outline

The present major Report is organized as follows:

Chapter 1: Introduction. In this chapter, we present the importance of functional requirements and non-functional requirements. Then the purpose and problem statement are outlined.

Chapter 2: Background. In this chapter, we introduce some backgrond knowledge including software development process, NFRs, software quality, SQA and V&V, etc. This chapter mainly presents backgrond knowledge necessary for this report.

Chapter 3: Functional Requirements. This chapter discussed functional requirements, including the category, methods used to deal with (collect, document, validate and verify) them, some support tools also discussed in this chapter.

Chapter 4: Non-Functional Reqirements. This chapter discussed non-functional requirements, including the category, the source of NFRs, methods to deal with(identify, document, validate and verify) them, some tools also discussed.

Chapter 5: Conclusion. This chapter conclude the work done by this report, and we present a further work to be done in this chapter.

Chapter 2 Background

2.1 Introduction

In this Chapter, the software development processes are introduced, and the role of both functional and non-functional requirements in the software development are outlined.

Software systems must be more than just a bunch of code modules. They must be structured in a way that enables scalability, security, and robust execution under stressful conditions. Large software projects have a huge probability of failure - in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than that it will succeed. Software Engineering is a discipline for the systematic construction and support of large software products so they can safely fill the uses to which they may be subjected. In the IEEE Standard Glossary of Software Engineering Terminology, the notion of software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [IEE90].

2.2 Software Development Process

Software development process is a coherent set of activities for software modeling, input and output (sometimes known as milestones) for each activity, and associated artifacts. A software process model is an abstract representation of a software process. Each process model represents a process from a particular perspective, and only provides partial information about that process. The oldest process model is the waterfall model. Recently, some new models such as Rational Unified Process, and eXtreme Program process were used more and more widely. A number of different general models of software development are listed below.

2.3 Waterfall Model

Waterfall model is the software development process model with longest history.

In 1970, Winston W. Royce introduced waterfall model in his paper [WIN70], and from then on, waterfall model has been used in software development. The process waterfall model includes five sequential activities: requirements definition, system and software design, implementing and unit testing, intergrate and system testing, operation and maintenance (see Figure 2-1).

Waterfall model is decomposed into phases, each phase corresponding to one activity and producing a delivery artifact.

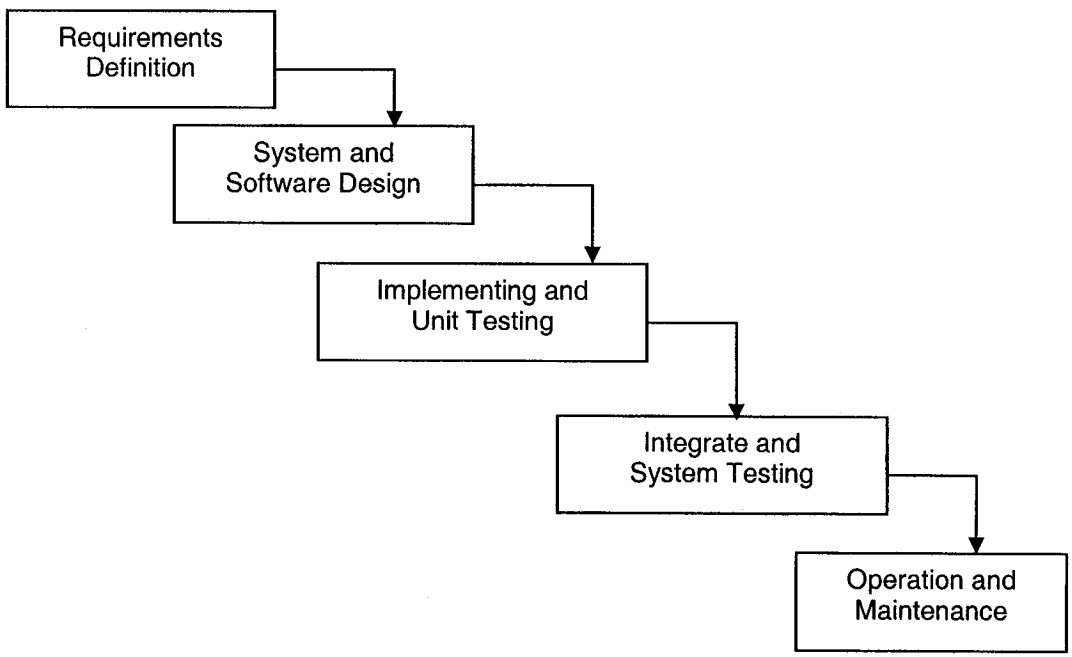

Figure 2-1: Waterfall Model

Phases In Requirements analysis and definition phase, the requirements are collected from the customers and defined, and the Software Requirements Specification (SRS) document is produced. SRS serves as a contract between the customers and the developers, therefore all the requirements defined should be understandable by both customer and system developers.

In System development process, requirements are refined into software and hardware requirements, the system architecture and system's components interfaces are fixed. The deliverable produced in this phase is the Design Document.

The system is realized as a set of components in Implementing and Unit Testing phase. Code is the main production of this phase. Units of code are individually tesed, and the test cases and test results are reported.

In Integrate and System Testing phase, units are integrated, integrate tests are performed and reported. If the testing results are satisfactory, the system is delived to the end-users.

In the last phase - Operating and Maintenance, bugs reported by the users are fixed and some enhancements are added to the software's functionality.

In the Waterfall Model the phases are sequential, that is, each phase (starting from the second) must begin after the previous phase has been ended. Waterfall Model is closer to the management ideas on software development organization, so it was enthusiastically welcomed by managers when it appeared due to its simplicity and clarity.

Applicability When the software requirements are clear from the beginning of the development process, Waterfall Model can be a very good chocie due to its transparency and ease of control. However, Waterfall Model is not feasible for the development of complex systems. According to the description above, Waterfall Model has a very late "product deployment", that is, we get the product to be delivered to users almost in the end of the process without giving any feedback on its development to the customers until the end of the process. If the user is not satisfied with the product, the whole development process has to be repeated.

2.3.1 Prototyping, Iterative and Incremental Models

Complex and evolving systems require a different approach to their development. Prototyping, Iterative and Incremental models have been created and used in software industry

for development of complex systems. Evolutionary Development Model and RUP (Rational Unified Process) are two of them.

Evolutionary Development Model is based on an idea of building a prototype and continually improving it according to customer feedbacks at each iteration. Users are glad to see the software continuously improving, so the model has been accepted quickly. In this model, there are two kinds of development:

- *Exploratory Programming*: Query customer requirements. First implement basic requirement and later add more complex ones.
- *Throw-away-Prototyping: Build prototype to understand the* customer's requirements. The prototype and experiments with the customer help to define the requirements.

The Figure 2-2 illustrates the Evolutionary Development Process:

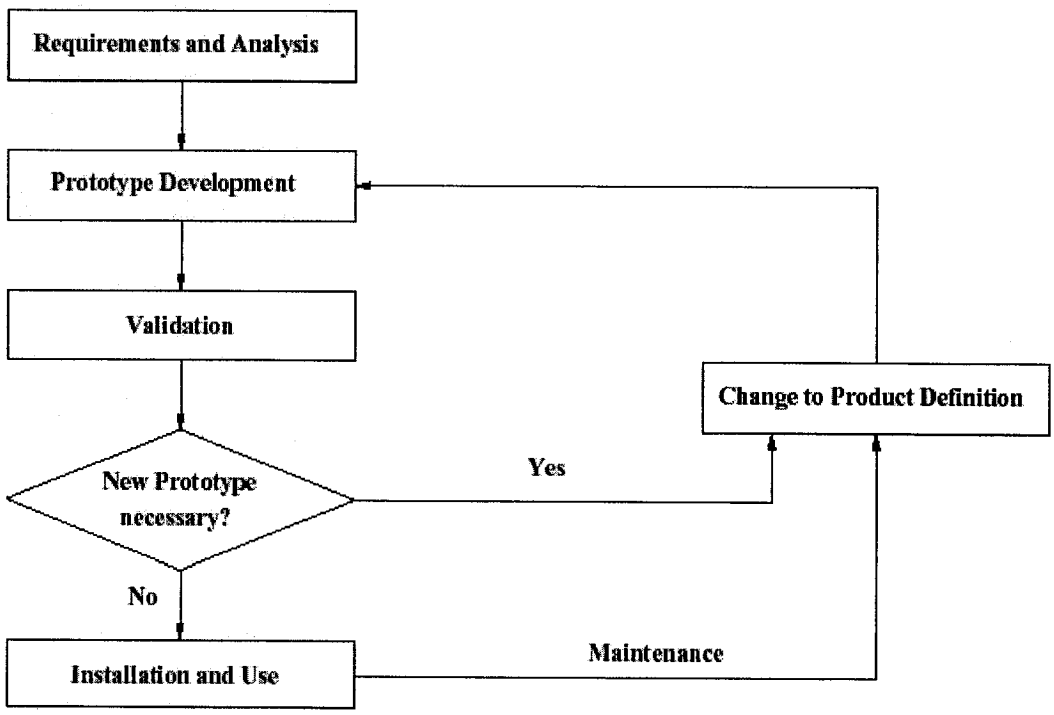

Figure 2-2: Evolutionary Development Process

Evolutionary Model is a theoretically model with wide applicability, but it still has some shortcoming, for example, the process is not transparent, the code would not be well structured due to the frequent modifications, and there is a need for skilled team to use this model.

Rational Unified Process (RUP) is another model used widely in the industry. RUP is first introduced by Rational [RAT00], this model can be integrated with Rational's tools and UML is used to model the system in the whole process. RUP is a configurable process. The Unified Process fits small development teams as well as large development organizations. RUP is founded on a simple and clear process architecture that provides commonality across a family of processes. Yet, it can be varied to accommodate different situations. It contains a Development Kit, providing support for configuring the process to suit the needs of a given organization. RUP can be cut down to customize the unique development process of specific software, with the reasonable cut down; we can use RUP as a sharp weapon.

2.3.2 UML™

The OMG's Unified Modeling Language™ (UML™)[UML98] helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. Using any one of the large number of UML-based tools on the market, you can analyze your future application's requirements and design a solution that meets them, representing the results using UML's twelve standard diagram types.

UML can be used so widely that you can model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network, in UML. Its flexibility lets you model distributed applications that use just about any middleware on the market. In fact, many extensions of UML have been produced that you can use UML to using UML in specific fields, for example, in embeded field and RT system Modeling. UML support program language widely. Using UML, you can ignore the difference from different languages, systems built upon one language can be a natual fit for other languages.

Also, UML can be used to do other useful things. For example, some tools analyze existing source code and reverse-engineer it into a set of UML diagrams. Another example: In spite of UML's focus on design rather than execution, some tools on the market execute UML models.

UML is useful to define requirement, either functional requirements or non-functional requirements. Functional requirements can be defined by Use Case Diagram; The ways to using UML to define non-functional requirements are also mentioned in some papers.

The most critical stage in the software development process is the requirements phase as errors at this stage inevitably lead to later problems in the system design and implementation. In general, requirements are partitioned into functional requirements and non-functional requirements. The rest of this Chapter is an overviews the role of the requirements (functional and non-functional) in the software development process.

2.4 Software Quality

Within an information system, software is a tool, and tools have to be selected for quality and for appropriateness. This section introduces software quality, including software quality models, framework and quality factors.

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular project, to be discussed and determined at the time that the product requirements are determined. Qualities may be present or absent, or may be matters of degree, with tradeoffs among them, with practicality and cost as major considerations. The software engineer has a responsibility to elicit the system's quality requirements that may not be explicit at the outset and to discuss their importance and the difficulty of attaining them.

Various researchers have produced models (usually taxonomic) of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of

software products. The models often include metrics to "measure" the degree of each quality attribute the product attains.

Usually these metrics may be applied at any of the product levels. They are not always direct measures of the quality characteristics of the finished product, but may be relevant to the achievement of overall quality. Some of the classical quality models are McCall, Boehm [BOE78], and others and are discussed in the texts of Pressman [PRE97], Pfleeger [PFL98] and Kan [KAN94]. Each model may have a different set of attributes at the highest level of the taxonomy, and selection of and definitions for the attributes at all levels may differ. The important point is that the system software requirements define the quality requirements and the definitions of the attributes for them.

2.4.1 Measuring the quantity of quality

A motivation behind a software project is a determination that it has a value, and this value may or not be quantified as a cost, but the customer will have some maximum cost in mind. Within that cost, the customer expects to attain the basic purpose of the software and may have some expectation of the necessary quality, or may not have thought through the quality issues or cost. The software engineer, in discussing software quality attributes and the processes necessary to assure them, should keep in mind the value of each one. A discussion of measuring cost and value of quality requirements can be found in [WEI93] Chapter 8 and [JON91] Chapter 5.

2.4.2 ISO 9126 Quality Description

Terminology for quality attributes differs from one model to another; each model may have different hierarchical levels and a different total number of attributes. One attempt to standardize terminology in an inclusive model resulted in ISO 9126 [IEE91]. ISO 9126 is concerned primarily with the definition of quality characteristics in the final product. ISO 9126 sets out six quality characteristics, each very broad in nature. They are divided into 21 attributes, or

subcharacteristics. Some terms for characteristics and their attributes are used differently in the other models mentioned above, but ISO 9126 has taken the various sets and arrangements of quality characteristics and has reached consensus for that model. Other models may have different definitions for the same attribute. A software engineer understands the underlying meanings of quality characteristics regardless of their names and their value to the system under development or maintenance.

2.4.3 Quality Attributes for the Engineering Process

Other considerations of software systems are known to affect the software engineering process while the system is being built and during its future evolution or modification, and these can be considered elements of product quality. These software qualities include, but are not limited to:

- Code and object reusability
- Traceability of requirements from code and test documentation and to code and test documentation from requirements
- Modularity of code and independence of modules.

These software quality attributes and their subjective or objective measurement are important in the development process, particularly in large software projects. They can also be important in maintenance (if code is traceable to requirements – and vice/versa, then modification for new requirements is facilitated). They can improve the quality of the process and of future products (code that is designed to be reusable, if it functions well, avoids rewriting which could introduce defects).

Table 2-1 is the software quality characteristics and Attributes from the ISO 9126 view.

Table 2-1: software quality characteristics and Attributes - the ISO 9126 view

Characteristic/ Attribute	Short Description of the Characteristics and the concerns Addressed by Attributes
Functionality	Characteristics relating to achievement of the basic purpose for which the software is being engineered
Suitability	The presence and appropriateness of a set of functions for specified tasks
Accuracy	The provision of right or agreed results or effects
Interoperability	Software's ability to interact with specified systems
Compliance	Adherence to application-related standards, conventions, regulations in laws and protocols
Security	Ability to prevent unauthorized access, whether accidental or deliberate, to programs and data.
Reliability	Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time
Maturity	Attributes of software that bear on the frequency of failure by faults in the software
Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure
Usability	Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users
Understandability	The effort required for a user to recognize the logical concept and its applicability
Learnability	The effort required for a user to learn its application, operation, input, and output
Operability	The ease of operation and control by users
Efficiency	Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions
Time behavior	The speed of response and processing times and throughput rates in performing its function
Resource behavior	The amount of resources used and the duration of such use in performing its function
Maintainability	Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functional specifications
Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified
Changeability	The effort needed for modification fault removal or for environmental change
Stability	The risk of unexpected effect of modifications
Testability	The effort needed for validating the modified software
Portability	Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another
Adaptability	The opportunity for its adaptation to different specified environments
Installability	The effort needed to install the software in a specified environment
Conformance	The extent to which it adheres to standards or conventions relating to portability
Replaceability	The opportunity and effort of using it in the place of other software in a particular environment

2.4.4 SQA and V&V

The set of requirements has a direct effect on the quality of other products, down to the delivered software. While the software engineering process builds quality into software products and prevents defects, the software engineering process also employs supporting processes to examine and assure software products for quality. The supporting processes conduct activities to ensure that the software engineering process required by the project is followed. Software Quality Assurance (SQA) and Validation and Verification (V&V) umbrella activities examine software through its development and maintenance, detect defects and provide visibility to the management in determining how well the software carries out the documented requirements.

The SQA role with respect to process is to ensure that planned processes are appropriate and have been implemented according to their plans and that relevant measurements about processes are provided to the appropriate organization.

The V&V process determines whether products of a given development or maintenance activity conform to the requirements of that activity and those imposed by previous activities, and whether the final software product satisfies its intended use and user needs. Verification ensures that the product is built correctly, that is, verification determines that software products of an activity fulfill requirements imposed on them in the previous activities. Validation ensures that the right product is built, that is, the final product fulfills its specific intended use. The activities of validation begin early in the development or maintenance process, as do those of verification. V&V provides an examination of every product relative both to its immediate predecessor and to the system requirements it must satisfy.

The visibility comes from the data and measurements produced through the performance of tasks to assess (examine and measure) the quality of the outputs of the software development and maintenance processes while they are developed.

Many SQA and V&V evaluation techniques may be employed by the software engineers who are building the product. And the techniques may be conducted in varying degrees of

independence from the development organization. Finally, the integrity level of the product may drive the degree of independence and the selection of techniques.

2.5 Role of Functional and Non-Functional Requirements in the Development Process

The major objective of the requirements phase is defining the purpose of a proposed system and outlining its external behavior. Requirements generally express what an application is meant to do. They do not attempt to express how to accomplish these functions. The set of requirements for the system should describe the functional and non-functional requirements so that they are understandable by system users who don't have detailed technical knowledge. Functional requirements are associated with specific functions, tasks or behaviours the system must support, while non-functional requirements are constraints on various attributes of these functions or tasks.

Functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform, for instance, UI requirements, database requirements are functional requirements. Generally, functional requirements are what we focus first. Traditional methods for requirements analysis often focus on functional requirements, interview is a common way to collect customers' requirements, from the interview, we can get function points by picking verbs and nouns, but we cannot recognize the hidden non-functional requirements from it. In the requirements definition phase, what we get from customers are almost functional requirements because customers always focus on "what we want" instead of "what quality the product should have", the other reason is customers always think software quality as hidden things they cannot controll.

With the need for software quality increased, non-functional requirements were paid more and more attention to. It can be helpful to think of non-functional requirements as adverbially related to tasks or functional requirements: how fast, how efficiently, how safely, etc.

The next two Chapters introduce the role of requirements in software development process, and survey the methods for specifying functional and non-functional requirements.

Chapter 3 Functional Requirements

3.1 Category

Functional requirements are always defined as what the system should really do. To give a accurate category of functional requirements is an impossible task because a software system is developed to satisfy some paticular requirements, which may be very difference from them of other customers.

However, we can assign a common category to the functional requirements. Let us think about a software system: firstly, every system should have a data process mechanism, so there must be some requirements on data processing, for example, how to store the data, how to structure the data and how to process the data. We can call all these requirements **"data requirements"**. Secondly, every system should have relationship with other systems, for example, with operation system, with some old systems already used, and/or with customers. These requirements can be categorized as **"interface requirements"**. Lastly, as the most important things, we must develop a system to help customers to solve problems in the real world, that is, we must first collect the requirements related to customers' problem. we can categorize these requirements **"operation requirements"**. "Operation requirements" are always most important to customers for the software system is used to solve their problem.

"Data requirements", "interface requirements" and "operation requirements" are not a standard way to cataloge funtional requirements, but in this report, we can use them to describe functional requirements.

3.2 Denotation of Functional Requirements

A functional requirement means a function the system should have. We can use different ways to record a functional requirement. Using natural language to denote functional requirements is a common way, an example of functional description always seems like this:

[Function]

Name, purpose

Preconditions

Description

End result

Exceptions

Describing functional requirements in natural language bring benefits: in requirements analysis phase, we use natural language (for example, English) to communicate with customers, so we can easily use the same way to record customers' requirements directly. A traditional way to collect customers' requirements is picking verbs and nouns from words of customers. Functional requirements thus can be generated from them. However, denoting functional requirements by natural language bring some trouble. As we all know, natural language is sometimes ambiguous, different people may not have same "feeling" when they read same words. If natural language was used as the only tool to record functional requirements, customers and requirements analysts may reach the seeming consistent but for the real consistent.

To deal with inconsistent requirements, many approaches to requirements analysis have been developed over the last few years. The different techniques are listed below.

3.3 Classification of Requirements and Analysis Techniques

According to a survey paper[GRO95], requirements and analysis techniques consist of four major classes and various subclasses. These major classed of techniques are as follows:

1. **Formal methods** are based on translation of requirements into mathematical form. Eight different techniques were discovered.

2. **Semi-formal** methods are based on the expression of requirement specifications in a special requirement language. Eleven different individual techniques were discovered.

3. **Review and Analysis (informal method)** are based on reviews by special personal of the adequacy of requirement specification according to a pre-established set of criteria and detailed checklists and procedures. Seven different methods were identified.

4. **Tracing and Analysis Techniques of requirements** are based on matching of each unique requirement element to design elements and then to the elements of the implementation. A typical tracing and analysis technique can be described as below: first, requirements analyzer go to interview customer, listen to their need, listen to their complaint, and listen to what they want, when he go back, he make requirements elements as baseline of the software. When the software is to be designed and implemented, all these requirement elements must be realized. Tracing is to validate. Traditionally, we collect functional requirements using these ways.

Formal methods are theoretically perfect methods. Theoretically, formal specification can lessen requirements errors by reducing ambiguity and imprecision and by clarifying instances of inconsistency and incompleteness. However, in the real world, it is difficult to use formal methods. First, the requirement shall be formulated unambiguously and "crystal clear". To ensure this, we must separate requirements into atomic and state them in terms of need, them we must find a way to make the definition of requirements unique and without different meaning. The other problem lies on formal language, there is still not a formal language accepted by all, so it's difficult

to communicate with each other using formal language, and, formal language is too complex to understand for most people.

PVS specification and verification system [SOW95][JCR95] is a formal specicfication and verification system commonly used in real-time control system. A project SafeFM [TBO94][SOW93] aimed to support the practical use of formal methods for high integrity systems not by producing new methods but by integrating formal methods into existing development and assessmant practice, demostrates an approach to formal methods of requirements analysis [BRU97].

Statechart [DHA87] is another effective method to deal with functional requirements, statechart mark requirement as node and then make chart to trace and validate them. Statechart make things easy to use chart instead of formal expressions.

CPN [MEF76] and RSML [WOL94] are also effective formal methods. By using these methods, requirement analysis can be more easy and the quality of requirements specification can be improved. However, the three methods described above have shortcoming – in some field, they can not be used due to their mathmetical nature.

SIS-RT [KUR97] is an improved method based on the three methods above, SIS-RT has three views; Inspection View, Traceability View, and Structure View. Inspection View of SIS-RT is designed to partially automate the software inspection process so that the burden of software inspection may be reduced. Requirement traceability analysis, which is considered as one of important activities of software V&V, is supported through Traceability View of SIS-RT. Also, Structure View of SIS-RT supports that the analyzer can easily specify a system using a formal specification method. By the three view, this method suggests an integrated approach with inspection and formal methods in order to support easy inspection and effective use of formal specification method.

Semi-formal methods are also introduced recently, they are based on special requirement languages, normally "Graphic" languages. GRA (Graphical Requirements Analysis) [NIH98] is a typical semi-formal method. GRA is a framework that describes a function into graphic logic

based with a structured form, by the predefined graphic symbols this framework can be used to analyze and design functional requirements. Graphical expressions can be understood more easily than formal language, so semi-formal methods are better way to deal with functional requirements. However, semi-formal methods are more difficult to master and when we translate textual requirements from customers to semi-formal, we may lose some information. Traditionally, and most common today, is that requirement specifications are written as text only, sometimes illustrated with figures.

Review and analysis methods may control functional requirements well. By the work of experts and customers, we can review requirements specification according to a pre-established checklist, thus we can find things wrong. But establishing an appropriate checklist is more difficult than it seems to be, considering the customers are not computer experts, perhaps the review should hardly reach the target. Tracing and analysis techniques can be useful in the development process, but it cannot tell if the requirements are what customers "exactly" want, and, it brings more additional work to the development process.

Use Cases [OBJ02] is another way to develop functional requirements. Intergrated with UML, Use Case is a powerful tool to develop functional requirements. Use case describe the senario users may perform and thus be more interesting to customers.

In [BUL96] a method for feature specification, design and validation is presented. This method uses USM (Use Case Map) [ISO89] for the design and documentation of features, uses LOTOS [AMY00] for the formal specification of features and for their formal verification. USM allows user to design senario visually, customers will master it soon and then use it to tell what they really want. LOTOS is an algebraic language with a history of applications to validation of distributed systems in general and to feature interaction detection in particular (see[KAM98], among an extensive literature).

A relational model for formal Object-Oriented requirements analysis is presented in [ZHI03]. This model uses use case and UML to analyze requirements and give a serial accurate mathmatical expressions.

As mentioned above, there is no perfect method that can be used in any case. Semi-formal methods are payed more and more attention because of their advantage, but traditional review and tracing methods are still useful in development process, especially when we decide to write requirement specifications mainly in textual mode.

3.4 Validation & Verification of Functional Requirements

Validation and Verification of functinal requirements are important work have to do. IEEE standards (IEEE 1012, IEEE 1012a, IEEE 1059) [IEE90] give guidelines in how to validate and verify the software. Normally, when we deal with functional requirements, we pay attention to the following:

- Does the requirement specification include all the functional requirements of customers?
- Does the requirement specification make thing clearly to all the relevant person, including costomers, designers, developers?
- How to verify the system implemented? How can we know wether the system is just the thing described in requirement specification?

Requirements validation is the process of establishing the adequacy, completeness and consistency of a requirements specification. There are five standard approaches to evaluate adequacy, completeness and consistency of a requirements specification regarding the users needs and expectations:

1. conducting reviews of the requirements specifications;

2. building and evaluating a prototype;

3. running a simulation;

4. using test and analysis functions intrgrated in specification tools/automated completeness and consistency analysis.

Test is the most common way to validate the functional requirements. System test is performaned to ensure all the functional requirements are implemented in the software system and no additional function is implemented. Test actives(unit test, integrated test and system test) must be preformaned in the whole development process to validate functional requirements. Different development processes have different test strategies, for instance, in trational Waterfall model, testing is the phase between coding phase and maintenance phase, but when we use XP process as our development process, we will find testing is among all the phases of the process. In this view, we can say different development process bring difference to functional requirements validation and verification.

Besides testing, many methods were introduced to deal with these problems. To validate the requirements, tarditional review and tracing methods are used, and, as new methods, GRA (Graphical Requirem nt Analysis) and methods using model are introduced recently. GRA uses a graphical semi-formal symbols to analyze and validate functional requirements while methods using model focus on the textual specification and use model to validate it [NIH98]. StateChart, CPN and RSML are all used as effective v&v harness [DHA87] [MEF76] [WOL94].

Function points review [MAU01] is a traditional method to validate and verify functional requirements. Requirements are numbered with a unique number so that they can be traced in the whole development process. In traditional development process, functional points review is a good method because functional requirements are not changed much in the whole development process, we can use requirements as the basis to check whether the system implement all the functional requirements. But function points review have its own shortcomings, the obvious one is we cannot prove the requirement specification is correct, all the thing it can prove is the system is.

V&V of functional requirements are related with process also. In [NUN02] an emergence-driven software process for agent-based simulation is presented; the process clarifies the traceability of micro and macro observations to micro and macro specifications in agent-based models. In this paper, the concept of hyperstructures was used to illustrate how micro and macro specifications interact in agent-based models.

Formalizing, in total or part, can be used to validate and verify functional requirements. Because of the complexity of the V&V process, the formal methods have to be supported by tools.

3.5 Tools

Many method metioned above are supported by tools, for example, SCR (Software Cost Reduction) tabular notation include a set of case tools to develop formal requirements specifications[NGL94]. Paper[NGL94] uses these tools to develop, verify and test formal functional specifications.

Now many tools were developed to help to analyze and validate requirements. Tools make requirements analysis process clear and under control. Commercial tools such as Rational Enterprise Suite has been used commonly in many company, it is powerful and suit for the team development, but it's expensive on the other hand. Some free tools which can be gotten with no charge are also useful and used by research institudes, schools and some company.

RAST(Requirement Analysis Support Tool) is presented by Seo-Young Noh, Shashi K. Gadia in their article [SEO99], this tool bases on linguistic information. By using RAST system, we can expect that requirement engineers can communicate with each other in common notations, and use logical expressions rather than using requirement specifications written in natural language.

Rational RequisitePro is another good requirements support tool, it can be integrated with other products of Rational and presents good flexiblity.

Each tool has its own benefits and certainly it's shortcoming. When we decide to choose one from them, the hardware support, the operation system ,the functions it presents and the price are to be considered. A detailed compare among the tools lists in table 3-1[IEE99].

In this Chapter, we have reviewed the role of functional requirements in the software development process, and have surveyed the existing methods for specifying, validating and

verifying the functional requirements. In the next Chapter, the role of the non-functional requirements and the methods for their specificaton are introduced.

Table 3-1: compare among the tools

Tool Name	Vendor	Description	Hardware Supported	Operating Systems
Caliber-RM	Technology Builders, Inc (TBI)	Requirements traceability tool	x86 Alpha MIPS	Win32
CORE	Vitech Corp.	Full life-cycle systems engineering CASE tool. It supports the systems engineering paradigm from the earliest days of concept development and proposal development, requirements management, behavior modeling, system design and verification process.	x86	Win16 Win32
CRADLE/REQ	3SL (Structured Software Systems)	Requirements Management tool capable of storing within its database, graphs, spreadsheets, tables, diagrams and any other information as part of a requirement.	SPARC x86 VAX RS6000	Win32 Unix
DOORS	Quality Systems & Software, Inc. (QSS)	Requirements traceability tool	SPARC x86 RS6000	Unix Win16 Win32 Solaris
icCONCEPT	Integrated Chipware	Requirements traceability tool. Replaces RTM	SPARC MIPS RS6000 x86 PA-RISC	Unix Win16 Win32
Life*CYCLE	Computer Resources International	Requirements traceability tool. No longer available		
QSSrequireit	Quality Systems & Software, Inc. (QSS)	Requirements trace tool that is integrated with Microsoft Word	x86	Win32
R-Trace	Protocol	Requirements traceability tool. No longer available	SPARC	Unix
RDD-100	Ascent Logic Corporation	Requirements and Simulation tool suite	x86 SPARC PPC	Unix Win32 MacOS
RDD-2000	Ascent Logic Corporation	Requirements and Simulation tool suite	x86 SPARC PPC	Unix Win32 MacOS
RDT	IGATECH Systems Pty Limited	Requirements traceability tool	x86	Win32

Tool Name	Vendor	Description	Hardware Supported	Operating Systems
RequisitePro	Rational Software	Requirements traceability tool	x86 Alpha MIPS	Win32
RTM	Integrated Chipware	Requirements traceability software. See icCONCEPT product.	SPARC MIPS RS6000 x86	Unix Win16 Win32
RTS	Electronic Warfare Associates, Inc.	Requirements Traceability Systems (RTS). Complete foundation for tracking the requirements of a software/hardware project through the accompanying documentation and source code. This includes tracking the development and testing status of requirements	SPARC	Unix
SLATE	SDRC - TD Technologies	System Level Automation Tool for Engineers (SLATE) is used to capture, organize, build and document system-level designs from raw concepts through structural partitioning. Interfaces to Office 97, Project and CASE tools.	SPARC x86 PA-RISC	Solaris WinNT HP-UX
Tofs	Tofs AB	Tool For Systems. Assists you in realizing and managing not only software, but also the manual (human) and hardware (electronic, hydraulic, mechanic, etc) parts of a system, which complete the system's missions together with the software.	x86	WinNT
Tracer	ISDE	Requirements traceability tool	x86	Win32 Win16
Vital Link	Compliance Automation Inc.	Requirements traceability tool	x86	Win32
XTie-RT	Teledyne Brown Engineering	Requirements traceability tool	x86(client) SPARC(server)	Win32 Unix

Chapter 4 Non-Functional Requirements

4.1 Introduction

According to IEEE standard [COM97] , non-functional requirement is defined as " a software requirement that describes not what the software will do, but how the software will do it, for example, software performance requirements, software design constraints, software external interface requirements and software quality attributes. Non-functional requirements are difficult to test; therefore, they are usually evaluated subjectively."

NFRs are described in [BOE78] as Interface requirements, Performance requirements, Operating requirements, Lifecycle requirements, Economic requirements and Political requirements; while in another view, NFRs can be classified as accessibility, adaptability, controllability, etc. Non-functional requirements are always related to software quality factors. Not long ago, Non-functional requirements had yet been neglected by the requirements engineering practice and research. But recently, with the need of building high-quality software growing, an increasing trend to build better software systems has highlighted the need to take non-functional requirements more seriously. Non-functional requirements were talked about by more and more people, many new framework and method to identify, control and manage NFRs were introduced, and many support tools were applied in the real development process.

From the SQA point of view, Non-functional requirements are usable descriptions of a software system. We can use NFRs framework to manage Non-functional requirements and thus improve the quality of the software system.

Non-functional requirements are hard to be measured directly. Whether or not a product meets its functional requirements is pretty straight forward. Either the product correctly responds to an input or it does not. Moreover, there are several notations for defining functional requirements precisely and unambiguously, to ensure that such requirements are testable. But to the NFRs, NFRs are implicit attribute of the system, we can not easily judge a system meet the

NFRs or not, for example, how can we say a system is "fast enough" or "robust enough"? An available way to make NFRs be measured is to use an approximate range to define the Non-functional requirement: for instance, we can define a Non-functional requirement like this: "The response time can not be increased more than 10% while the system used by 200 users than by 1 user". That means you must know and document the response time of the situation when only 1 user use this system, than you can use this as the basis to check if the specific Non-functional requirement is met. Another way is QFD [LCH94] (Quality-Function Deployment) method to relate an immeasurable or hard-to-measure NFRs to one or more functional requirements.

Many Non-functional requirements conflict with one another. This conflict means it is hard to build a product that maximizes both attributes. When this happens, we can use priority to help us make decisions. Different Non-functional requirements have the different importance to end-users, so we can rank them by priority. When we have two or more attribute conflicts, we can choose the attribute that is prior to all others.

4.2 Sources for NFRs

NFRs (Non-functional requirements) are always related to software quality and are among the most expensive and difficult to deal with [BRO87] [DAV93] [BRE99][CYS01]. In real world, NFRs arise from the operating environment, the customers, and competitive products [RUT01]. We can classify the sources for NFRs as follows:

- **System Constraints:**

Software system always depends on some constraints, for instance, the operation system, the hardware on which system must depends, the development language system must use. All these constraints should be built into software system and thus constraints are origin of NFRs [MAL01].

- **User Objectives, Values and Concerns:**

In establishing run-time qualities for a system, it is important to identify all users (including other systems) that will interact with the system, and understand what quality attributes they care about. Sometimes a quality attribute may be a concern to one user, but it maybe a value for the other. Therefore, it is useful to perform direct elicitation of the values and concerns for customers [MAL01].

- **Development Organization Constraints:**

In product development, constraints placed by upper levels of management typically take the form of required time-to-market of the application or release and/or fixed development resources. When both variables are fixed, the feature requirements have to be strictly scoped. This shows up in what functionality is scoped for the current release and what is deferred, and in driving trade-offs among the quality attributes of the system. Other factors of the development organization, such as the background and skill-set of the engineers, may also place constraints on what the development organization can accomplish especially given other constraints like time-to-market.

- **Development Organization Objectives, Values and Concerns:**

Stakeholders in the development organization include strategic management (e.g., general manager and R&D/IT manager), program and project managers, architects, developers, quality assurance (testers), marketing and manufacturing engineers, etc. Their objectives, values and concerns may relate to the business performance, schedules, productivity and effectiveness, work-life balance, etc. For example, strategic management establishes the product portfolio plan, including planned releases (which products in what timeframe). The architects and technical managers may translate those portfolio objectives into development-time quality requirements such as extensibility, evolvability, and reusability [MAL01], knowing that the portfolio cannot be accomplished without these characteristics. Developers may be concerned that reuse artifacts in fact deliver the qualities their particular product requires. And so forth.

- **Competitors and Industry Trends:**

Benchmarks of competitors' processes (e.g., how many products they release per year, with how many people) and industry trends, may drive the organization to adopt more aggressive productivity objectives which may in turn translate into development-time qualities such as evolvability and reuse.

Once you have worked with stakeholders to gather their requirements, these needs to be documented in such a way that the architects, designers and implementers can all understand them and create a system that fulfills the requirements.

4.3 NFRs and Quality Models

When we say NFRs, what we mean is what quality targets should the system achieve, so we depend on software quality models to describe NFRs. Software quality models are used to determine to what extent software components satisfy the requirements of a given context of use.

Some software-centered quality standards have been proposed [ISO99][IEE92][ROM85]. Although each of them has its own specifities, some guidelines are common: a framework for the whole quality assessment process exist; software quality characteristics are identified and defined in a hierarchical manner; etc. A set of ISO/IEC standards are related to software quality, being standards number 9126 (which is in process of substitution by 9126-1, 9126-2 and 9126-3), 14598-1 and 14598-4 the more relevant ones [ISO99] which we have discussed in details in section 2.3. The main idea behind these standards is the definition of a quality model and its use as a framework for software evaluation. A quality model is defined by means of general characteristics of software, which is further refined into subcharacteristics in a multilevel hierarchy; at the bottom of the hierarchy appear measurable software attributes. Quality requirements may be defined as restrictions over the quality model.

The ISO/IEC 9126 standard states that characteristics at the top of the hierarchy are: functionality, reliability, usability, efficiency, maintainability and portability. These characteristics can be separated into accuracy, compliance, interoperability, security, suitability, faults, maturity, recoverability, learnability, operability, understandability, analysability, changeability, stability,

testability, adaptability, installability, replaceability. A detailed description of this is demostrated by Table2-1 in the Chapter2.

We can classify NFRs into three classes:

- **Product-Oriented Attributes**: Performance, Useability, Efficiency, Reliability, Security, Robustness, Adaptability, Scalability, Cost;
- **Family-Oriented Attributes**: Portability, Modifiability, Reusability;
- **Process-Oriented Attributes:** Maintainability, Readability, Testability, Understandability, Integratability, Complexity.

The reason why we classify these NFR attributes is that different roles related with the software, such as software developers or customers, concern different class. Software developers tend to pay attention to family-oriented and process-oriented attributes. These attributes have noting to do with saving money during development, testing, and maintenance. While customers may only pay attention to product-oriented attributes, because they just want a product which is cheap, robust, efficient, easy to install and operate. The maintainancibility, reusability and etc. normally are not the factors they concern.

4.4 Dealing with Non-Functional Requirements

When surveying existing approaches to NFRs, we distinguish them according to their support for different tasks: elicitation, documentation, and architecture alignment.

4.4.1 Elicitation

An important step in achieving successful software requirements and products is to achieve the right balance of quality attributes requirements. This involves identifying the conflicts among several desired quality attributes, and working out a satisfactory balance of attributes satisfaction.

Elicitation covers the following questions:

1. How to identify NFRs;
2. How to ensure consensus of all stakeholders (resolving conflicts among NFRs);
3. How to relate the NFRs, the FRs, and the architecture;
4. Major techniques for dealing with these questions are the followings;
5. Negotiation techniques. Examples for this technique are WinWin and structured client reviews (SCRAM);
6. Decomposition.

4.4.1.1 Negotiation techniques

- ***USC-CSE WinWin system***

The WinWin system[BOE94] is based on the win-win spiral model (Figure 4-1), which uses the Theory W win-win approach to converge on a system's next level objectives, constraints, and alternatives.

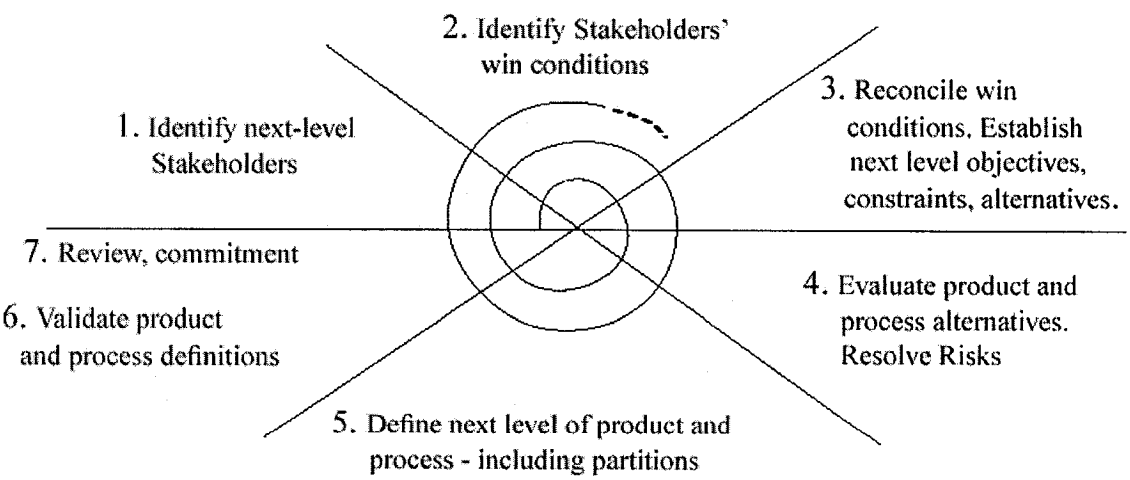

Figure 4-1: The win-win spiral model

Figure 4-2 shows the negotiation model used by WinWin, in terms of its primary artifacts and the relationships between them. Stakeholders begin by entering their Win Conditions, using a schema provided by the WinWin system. If a conflict among stakeholders' Win Conditions, an issue schema is composed, summarizing the conflict and the Win Conditions it involves.

For each issue, stakeholders prepare candidate Option schemas addressing the issue. Stakeholders then evaluate the Options, iterate some, agree to reject others, and ultimately converge on a mutually satisfactory option. The adoption of this option is formally proposed and ratified by an agreement schema, including a check to ensure that the stakeholders' iterated win conditions are indeed covered by the agreement.

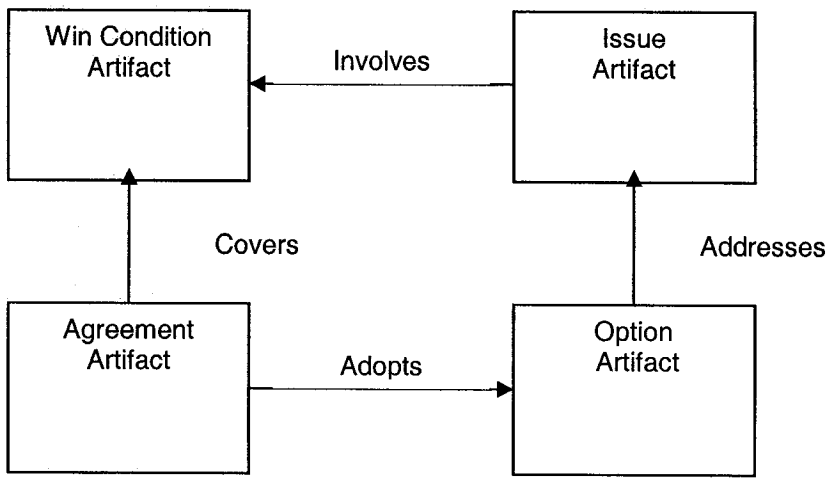

Figure 4-2: Win-Win Negotiation Model

In large systems involoving several dozens or more win conditions, it becomes difficult to identify conflicts among them becausae it is lack of a common domain model which provides a consistent semantic framework for all win conditions.The WinWin system provides us a model that acts to facilitate understanding of one's own and others' win conditions, and also of the relations among the win conditions. Understanding the win conditions in terms of domain functions and attributes, is a major facilitator for determining plausible conflicts. Figure 4-3 shows

part of the WinWin software engineering environment (SEE) domain taxonomy. The system enables stakeholders to negotiate common terminology as well as system requirements.

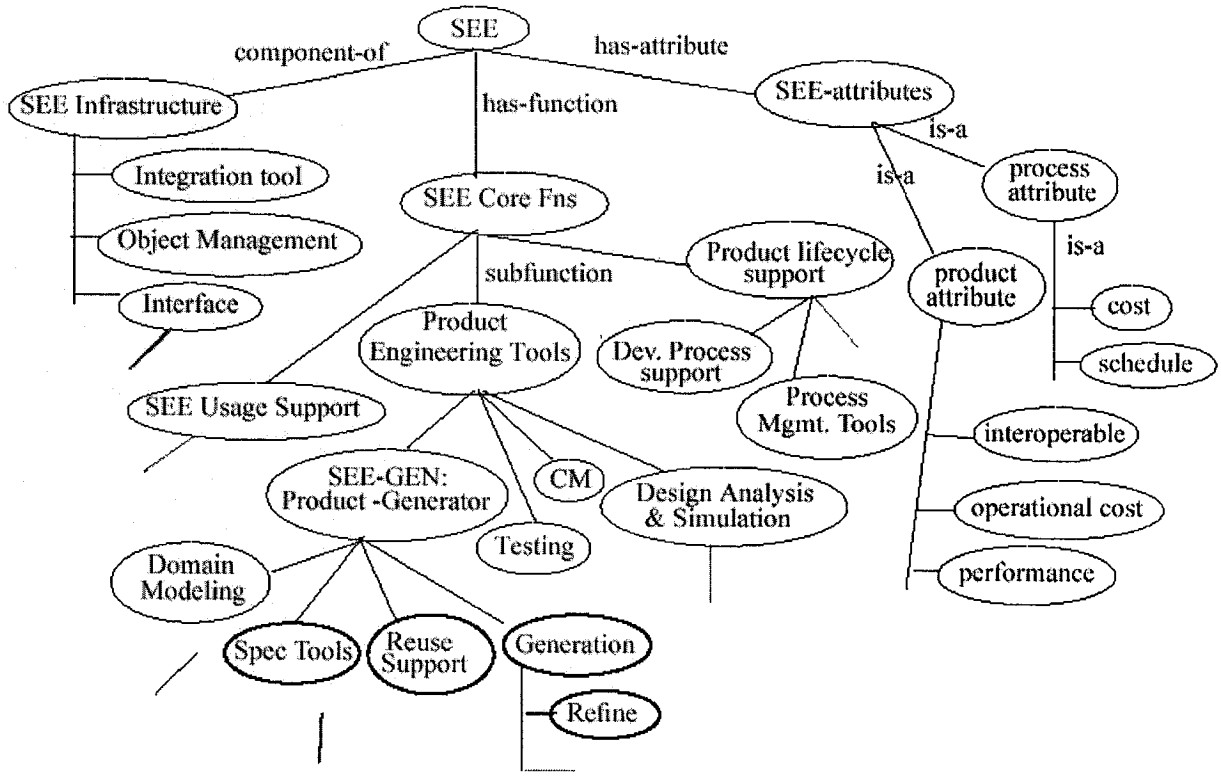

Figure 4-3: A portion of the SEE domain taxonomy

(The overall domain model has two components: a) a terminology database (not shown) that provides a description of all key terms that are used in defining a win condition; b) the domain taxonomy: a structured hierarchy of the domain entities and their attributes in the domain as well as their process specific attributes.)

4.4.1.2 Decomposition Technique

Decomposition encompasses the refinement of NFRs into more detailed NFRs. A very important aspect of non-functional requirements decomposition using the NFR Framework[CHU00] is that, as far as NFR softgoal are refined into more detailed ones, it is possible to identify interactions between non-functional requirements. These interactions include positive and negative contributions and have a critical impact on the decision process for achieving other non-functional requirements. A suitable way to deal with such complex

interdependencies is to assign priorities to non-functional requirements in order to make appropriate tradeoffs among NFRs.

For instance, Figure 4-4 shows a decomposition of non-functional requirements using the NFR Framework. The goal security of information is decomposed into the subgoals integrity, availability, confidentiality through an AND type of contribution (i.e. only if all subgoals are met the overall goal is achieved). While the goal system performance is decomposed into throughput and response time. Interestingly, it is necessary to address interactions between different kinds of non-functional requirements even though the non-functional requirements were initially stated as separate requirements. Note that cryptography contributes negatively (show as "-") for system performance. Since the NFR Framework facilitates the understanding of what a particular non-functional requirement means.

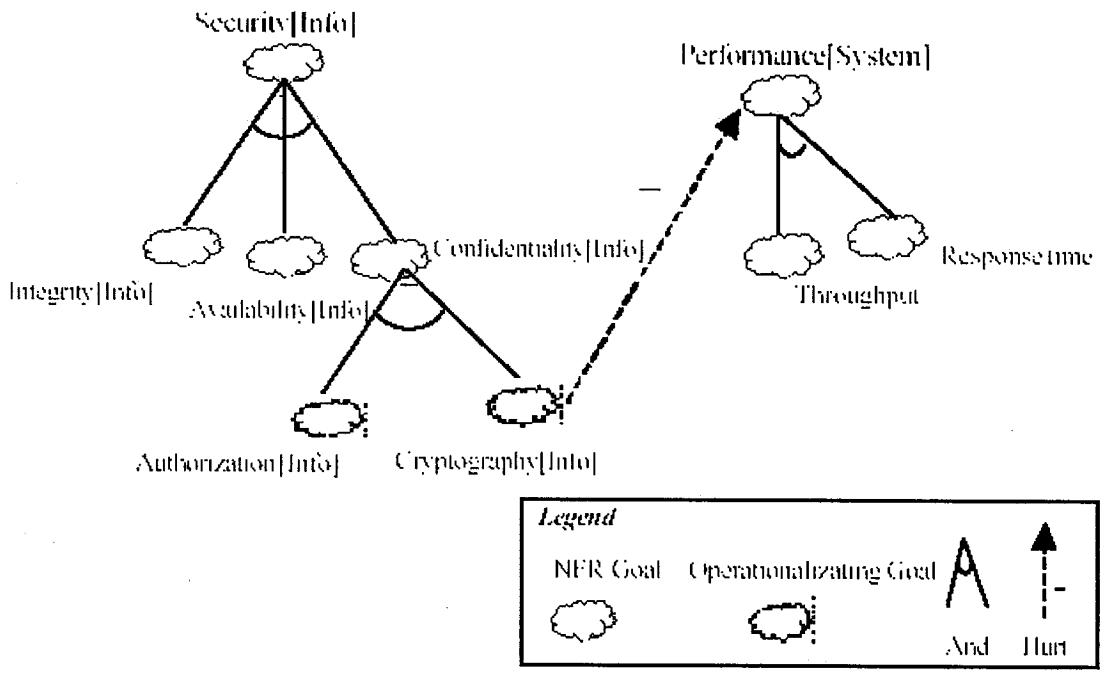

Figure 4-4: Decomposition of non-functional requirements using the NFR Framework

4.4.1.3 Operationalization Technique

Formally speaking operationalization is a process that maps declarative property specifications to operational specifications satisfying them.[EMM02] The operationalization can be represented in two different ways. The fist one is called dynamic operationalization. This type of operationalization can be faced as those that ask for abstract concepts and usually calls for some actions to be carried out. The second one is called static operationalization and usually expresses the need for some data to be used in the design of the software to store information that is necessary to satisfy NFRs[CYS01].

4.4.2 Documentation

We have realized the importance of NFRs, in order to deal with and build them into software; we have to document them first. To document NFRs, we have to solve at least two problem: "How to describe NFRs" and "Which additional information is necessary to deal with NFRs". Article [MRB95] give an answer to the former, it distinguish different facts on how to describe NFRs, namely ***concerns, system*** and ***environmental properties*** relevant for the NFR.

In [MRB95], each NFR attribute (performance, dependability, security and safety) should be identified with taxonomy (See Figure 4-5):

◆ **Concerns** — the parameters by which the attributes of a system are judged, specified and measured. Requirements are expressed in terms of concerns.

◆ **Attribute-specific factors** —properties of the system (such as policies and mechanisms built into the system) and its environment that have an impact on the concerns. Depending on the attribute, the attribute-specific factors are internal or external properties affecting the concerns. Factors might not be independent and might have cause/effect relationships. Factors and their relationships would be included in the system's architecture:

◆ **Performance factors** — the aspects of the system that contribute to performance. These include the demands from the environment and the system responses to these demands.

◆ **Dependability impairments** — the aspects of the system that contribute to dependability. There is a causal chain between faults inside the system and failures observed in the environment. Faults cause errors; an error is a system state that might lead to failure if not corrected.

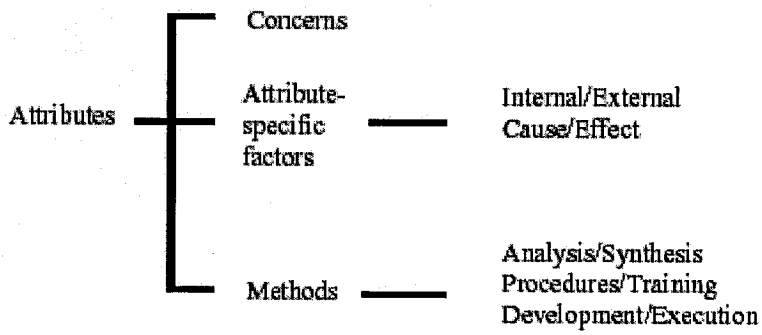

Figure 4-5: Generic Taxonomy for NFR

◆ **Security factors** — the aspects of the system that contribute to security. These include system/environment interface features and internal features such as kernelization.

◆ **Safety impairments** — the aspects of the system that contribute to safety. Hazards are conditions or system states that can lead to a mishap or accident. Mishaps are unplanned events with undesirable consequences.

◆ **Methods** — how we address the concerns: analysis and synthesis processes during the development of the system, and procedures and training for users and operators. Methods can be for analysis and synthesis, procedures and training, or procedures used at development or execution time.

Scenarios have been advocated as an effective means of acquiring and validating requirements as they capture examples and real world experiences that users can understand [CPO94]. Figure 6 illustrates the whole process of a Scenarios Method.

Paper [ASU98] proposes an analysis method that describes scenario templates for NFRs, with heuristics for scenario generation, elaboration and validation. The method contains a template for each high level NFR with embedded heuristics for scenario creation, validation and benchmark assessment by metrics. The templates give process guidance as well as scenario knowledge representation schemas. Templates for each high level type of NFR describe the necessary contents of a scenario that may be either generated or captured, an example scenario, validation guidelines for using the scenario in conjunction with other representations, and metrics for quality assessment. As many NFRs interact, choosing one solution may frequently adversely affect another property. To help tracing possible knock-on effects the method provides a comparison matrix as an aid memoir to trace interactions between NFRs. When interactions are found, these can be reflected in scenarios as a record of the problem or entered into the design rationale as synergistic or conflicting criteria.

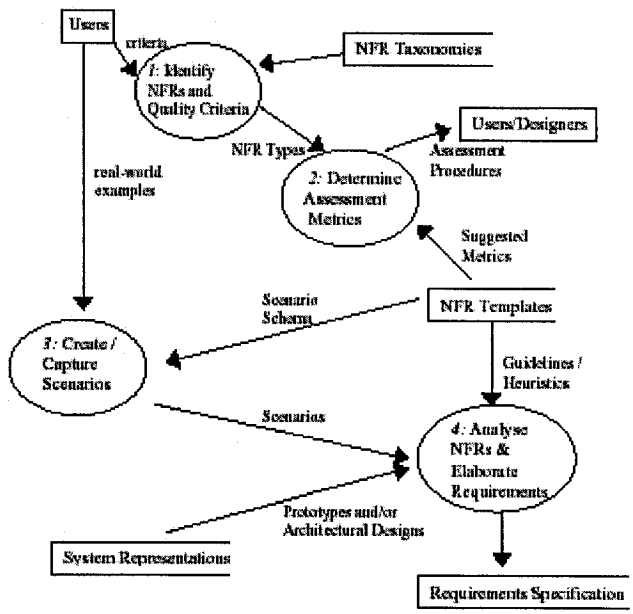

Figure 4-6: NFR Analysis Method

4.4.3 Architecture Alignment

When select a architecture, we have to take NFRs into acount. But how can we do this? Article [RKA99] presents a model named ATAM (Architecture Tradeoff Analysis Method) to provide a systematic method of evaluating scenarios against an architecture. ATAM have six steps to select architecture: Scenario brainstorming, Architecture presentation, Scenario coverage checking, Scenario grouping and prioritization, Map high priority scenarios onto architecture and Perform quality attribute-specific analysis. During these steps, the analysts may discover that the existing architecture is inadequate. As a consequence, architectural alternatives may be suggested, and these will such as: more architectural information, scenarios, environmental information, platform information, details about constraints, or justification for requirements.

Paper [DGR00] describe the way to using ***Design Partner*** in architecture alignment. Design Partner is now being used by many project and proved to be a useful tool to reuse and build a stable system. Paper [DGR00] presents the way in how to use ***Design Partner*** to select architecture while taking NFRs into account.

4.5 NFR Framework

NFR Framework is the first technique developed at the University of Toronto[LCH95][LC95][JMY92] to represent component-offered QoS (Quality of Service) contracts. NFR Framework and its associated method help the service developer to reason about the relative merits of one design solution over another with respect to their impact on particular QoS parameters and, also to reason about conflicts and synergies amongst QoS parameters themselves (e.g., the frequent conflict between space and time performance). In summary, the NFR Framework and its associated method helps designers reason about the tradeoffs between NFRs (including non-functional QoS parameters) when selecting amongst or combining design solutions.

NFR Framework can help to detect the architecture weakness, give the reason of weakness and guide to chage the architecture to improve the quality of software. After NFR Frameworks ware introduced in 1995, many new methods which depended on NFR Framework were developed, some of them are **Process-Oriented** while other are **Goal-Oriented**. Process-Oriented methods focus on process and easy to intergrated into develope process, while Goal-Oriented ones uses NFRs as goal to guide the refinition of reflect process.

Normally, NFR frameworks use priority to evaluate the architectures, different framework use different factors and metrics. Generally, NFR Framework requires the following interleaving tasks, which are iterative:

1. Develop the NFR goals and their decomposition;
2. Develop architectural alternatives;
3. Develop design tradeoffs and rationale;
4. Develop goal criticalities;
5. Evaluation and Selection.

Lawrence Chung and Nary Subramanian present a framework POMSAA (Process-Oriented Metrics for Software Architecture Adaptability) [LCH01] to provide numeric scores representing the adaptability of a software architecture as well as the intuitions behind these scores. In POMSAA framework, the intuitions behind the architectural adaptability scores are traced back to the "whys" of the architecture, namely, the requirements for which the architecture exists in the first place. POMSAA adds a metrification step to the NFR Framework and applies it at the architecture level. In the metrification step, either an automatic algorithm or a semi-automatic procedure is used to propagate metric values up the SIG to the NFR whose metric is to be computed.

Paper[NIX00] presents a performance requirements framework (PeRF). The purpose of PeRF is to deal with performance requirements of Information System, mainly the management of performace requirements. PeRF integrates and catalogues a variety of kinds of knowledge of

information systems and performance. These include: performance concepts, software performance engineering principles for building performance into systems, and information systems development knowledge. All this knowledge is represented using NFR framework.

A method named SA3 on generating adaptable UI is presented in [NAR99]. UI of a software system is treated as a software system itself in [NAR99], then SA3 is presented to use the principles behind the NFR Framework, particularly the latter's knowledge base properties, to automatically generate adaptable architectures, which can then be completed by the UI developer.

The NFR framework was one significant step in making the relationships between quality requirements and intended decisions explicit. The framework uses non-functional requirements to drive design to support architectural design, and to deal with change. Article [DAN99] presented a way to design using patterns. In this article the authors proved it is possible to use patterns to design software system satisifing NFRs.

4.6 NFRs and Software Architecture

NFRs are suggested to be built into software architecture [LCH00]. NFR framework presents ways to reflect NFRs into components of software architecture.

Paper [FRA98] presents an approach for incorporating non-functional information of software system into software architectures. In that paper, components present two distinguished slots: their non-functional specification, where non-functional requirements on components are placed, and their non-functional behavior with respect to these requirements. And, connector protocols may describe which non-functional aspects are relevant to component connections. A notation to describe non-functionality in a systematic manner was used to analyze aspects was described also.

Achieving an acceptable architecture requires an iterative derivation and evaluation process that allows refinement based on a series of tradeoffs. Researchers at the University of Texas at Austin are developing a suite of processes and supporting tools to guide architecture

derivation from requirements acquisition through system design. The various types of decisions needed for concurrent derivation and evaluation demand a synthesis of evaluation techniques, because no single technique is suitable for all concerns of interest. Two tools (RARE and ARCADE) can deal with NFRs were presented in paper [BAR02]. RARE guides derivation by employing heuristics knowledge base, and evaluates the resulting architecture by applying static property evaluation based on structural metrics. ARCADE provides dynamic property evaluation leveraging simulation and model checking.

Paper [SMI97] presents an architecturally based software reliability model and underlines its benefits. The model is based on an architecture derived from the requirements including both functional and non-functional requirements and on a generic classification of functions, attributes. The model accounts explicitly for the type of software development life cycle. It can incorporate any type of prior information, such as results of developers' testing or historical information on a specific functionality and its attributes, and is ideally suited for reusable software. By building architecture and deriving its potential failure modes, the model forces early appraisal and understanding of the weaknesses of the software, allows reliability analysis of the structure of the system, and provides assessments at a functional level as well as at the system level.

Also, UML is used to reflect NFRs into software architecture. MoFoV is presented by paper [BEN02], which is a support method for the design of complex systems. MoFoV is devoted to a functional view of systems. Assigned to technological contexts, this method is based on two generic models (which vary depending on each specific application domain). The first (MOGET) is dedicated to systems treatments (in terms of functions) whilst the second (MOGEP) deals with the potential properties of those systems.

In this Chapter, the role of non-requirements in software development process has been outlined, and the methods and tools for specifying non-functional requirements (NFR) have beed surveyed. The conclusions of this report are summarized in the following Chapter.

Chapter 5 Conclusions

5.1 Conclusions

This Major Report presents a survey on functional requirements and non-functional requirements. The functional and non-functional requirements play important role in software development. The research in the area of functional requirements is extensive, and recently the non-functional requirements (NFRs) have been paid more and more attention due to the higher software quality expectations of the customers.

The concepts of functional and non-functional requirements are introduced, methods to deal with them are also described, and tools for requirements engineering were compared.

The problems of functional and non-functional requirements lay on one of the following aspects: *identify*, *document*, *validate* and *verify*. In this report, we introduce the identify, document, validate and verify of functional requirements. As to NFRs, we introduce methods to identify and document NFRs, and the ways to deal with NFRs. The future research work in the area of NFRs has to be concerned with their validation and verification.

Chapter 6 References

6.1 References

[AMY00] Amyot, D., and Logrippo, L. "Use Case Maps and LOTOS for the Prototyping and Validation of a Mobile Group Call System". To appear in Computer Communications, 23(8),2000.

[ASU98] A. Sutcliffe and S. Minocha, "Scenario-based Analysis of Non-Functional Requirements", REFSQ'98, 1998.

[BAR00] Barbara Paech, Allen H. Dutoit, Daniel Kerkow, Antje von Knethen, Functional requirements, non-functional requirements, and architecture should not be separated–A position paper, 2000.

[BAR02] Barber K.S., Graser T., Holt J., Enabling iterative software architecture derivation using early non-functional property evaluation. IEEE International Conference on Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th, P 172 – 182.

[BAS98] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

[BEN02] Benaben F., Antoine C., Pignon, J.-P., A UML-based complex system design method MoFoV (Modeling/Formalizing/Verifying) Systems, 2002 IEEE International Conference on Man and Cybernetics, Volume 3 ,P 6 pp. vol.3

[BOE78] Boehm, B. W. et al, Characteristics of Software Quality, TRW series on Software Technologies, Vol. 1, North Holland, 1978

[BOE94] Boehm, B., Bose, P., Horowitz, E., and Lee, M., "Software Requirements As Negotiated Win Conditions", Proceedings of the First International Conference on Requirements Engineering (ICRE94), IEEE Computer Society Press, Colorado Springs Colorado, April 1994, pp. 74-83

[BRE99] Breitman,Karin Koogan, Leite J.C.S.P. e Finkelstein Anthony. The World's Stage: A Survey on Requirements Engineering Using a Real-Life Case Study. Journal of the Brazilian Computer Society No 1 Vol. 6 Jul. 1999 pp:13:37.

[BRO87] Brooks Jr.,F.P."No Silver Bullet: Essences and Accidents of Software Engineering" IEEE Computer Apr 1987, No 4 pp:10-19, 1987.

[BRU97] Bruno Dutertre and Victoria Stavridou, "Formal Requirements Analysis of an Avionics Control System", IEEE Transactions on Software Engineering, Vol. 23. No. 5. 1997

[BUL96] Buhr, R.J.A. and Casselman, R.S. Use Case Maps for Object-Oriented Systems, Prentice-Hall, 1996.

[CHU00] Chung, L., Nixon, B., Yu, E., and Mylopoulos, J., Non-Functional Requirements in Software Engineering. Kluwer Academic Publisher, 2000.

[COM97] Computer Industry Daily reported on a Sequent Computer Systems, IEEE, December 1997

[CPO94] C. Potts, K. Takahashi and A. I. Anton, 'Inquiry-based Requirements Analysis', IEEE Software, vol. 11, no. 2, pp. 21-32, 1994.

[CYS01] Cysneiros,L.M., Leite, J.C.S.P. and Neto, J.S.M. "A Framework for Integrating Non-Functional Requirements into Conceptual Models" Requirements Engineering Journal – Vol 6 , Issue 2 Apr. 2001, pp:97-115.

[DAN99] Daniel Gross, Eric Yu, From Non-Functional Requirements to Design through Patterns, 1999.

[DAV93] Davis, A. "Software Requirements: Objects Functions and States" Prentice Hall, 1993.

[DGR00] D. Gross, E. Yu, "From Non-functional requirements to design through patterns", REFSQ'00, pp.86-97, 2000.

[DHA87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming, vol. 8, pp.231-274, 1987.

[EMM02] Emmanuel Letier and Axel van Lamsweerde, Deriving Operational Software Specifications from System Goals, 2002

[FRA98] Franch X., Botella P., Putting non-functional requirements into software architecture, Ninth International Workshop on Software Specification and Design, 1998. Proceedings, P 60 – 67.

[GRE98] Gready Booch, James Rumbaugh, Ivar Jacobson. 1998. The Unified Modeling Language User Guide

[GRO95] Groundwater E.H., Miller L.A., Mirsky S.M. 1995. Guidelines for the Verification and Validation of Expert System Softwre and Conventional Software. Survey and Document of Expert System Verification and Validation Methodologies, NUREG/CR-6316, SAIC-95/1028. Vol.1-7

[IEE90] IEEE Std. 610.12-1990. IEEE Standards Glossary of Software Engineering Standards.

[IEE91] IEEE standard: Quality Characteristics and Guidelines for their Use, 1991

[IEE92] IEEE Computer Society. IEEE Standard for a Software Quality Metrics Methodology. IEEE Std. 1061-1992, New York, 1992.

[IEE99] IEEE-1220 SE Tools Taxonomy - Requirements Analysis Tools, International Council on Systems Engineering, 1999,

[ISO89] ISO, Information Processing Systems, Open Systems Interconnection, LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807, Geneva, Switzerland, 1989 (E. Brinksma, Ed.).

[ISO99] ISO/IEC Standards 9126 (Information Technology – Software Product Evaluation –Quality Characteristics and Guidelines for their use, 1991) and 14598 (Information Technology – Software Product Evaluation: Part 1, General Overview; Part 4, Process for Acquirers; 1999).

[JCR95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A tutorial introduction to PVS" in WIFT95 Workshop on Industrial-Strength Formal Specification Techniques, April 1995

[JMY92] J. Mylopoulos, L. Chung and B. Nixon, Representing and using non-functional requirements: a process-oriented approach. IEEE TSE, Vol. 18, June 1992.

[JON91] Jones, Capser, Applied Software Measurement, McGraw-Hill, Inc., 1991

[KAM98] Kamoun, J. and Logrippo, L. "Goal-Oriented Feature Interaction Detection in the Intelligent Network Model". In K. Kimbler and L. G. Bouma (Eds), Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98), Lund, Sweden, Sept. 1998. IOS Press, 172-186.

[KAN94] Kan, Stephen, H., Metrics and Models in Software Quality Engineering, Addison-Wesley Publishing Co., 1994

[KUR97] Kurt Jensen, "Coloured Petri Nets (Basic Concepts, Analysis Methods and Practical Use Volume 1), Second Edition", Springer-Verlag Berlin Heidelberg, 1997.

[LC95] L. Chung, B.A. Nixon and E. Yu, Using non-functional requirements to systematically support change. Proceedings Requirements Engineering 1995.

[LCH00] L. Chung, B.A. Nixon, E. Yu, and Mylopoulos,J. "Non-Functional Requirements in Software Engineering", Kluwer Academic Publishers 2000.

[LCH01] L. Chung, Nary Subramanian, Process-Oriented Metrics for Software Architecture Adaptability, Proceedings of the Fifth International Symposium on Requirements Engineering (RE'01), 2001.

[LCH94] L. Chung, Brian A. Nixon and Eric Yu, "Using Quality Requirements to systematically develop quality software", fourth international conference on software quality, 1994.

[LCH95] L. Chung and B.A. Nixon, Dealing with non-functional requirements: three experimental studies of a process oriented approach. Proceedings of the 17th ICSE, pp. 25-37, 1995.

[LUI02] Luiz Marcio Cysneiros Julio César Sampaio do Prado Leite, Using the Language Extended Lexicon to Support Non-Functional Requirements Elicitation, 2002.

[MAL01] Malan, R. and D. Bredemeyer, "Functional Requirements and Use Cases", October 2001.

[MAU01] Mauricio Aguiar, Caixa Economica Federal ,"Applying Function Point Analysis to Requirements Completeness", The Journal of Defense Software Engineering, Feb 2001 Issue

[MEF76] M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM system Journal, Vol. 15, No. 3, pp. 182-211, 1976.

[MRB95] M. R. Barbacci, M. H. Klein, T. Longstaff and C. Weinstock, "Quality Attributes", Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, December 1995.

[NAR99] Nary Subramanian, Lawrence Chung, Adaptable User Interface Generation, 1999.

[NGL94] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," IEEE Transaction on Software Engineering, vol.20, no.9, sept. 1994.

[NIH98] Nihal Kececi, Wolgang A. Halang, Alain Abran. 1998. A Semi-formal method to verify correctness of functional requirements specifications of cmplex embedded system

[NIX00] Nixon, B.A., Management of performance requirements for Information systems, IEEE Transactions on Software Engineering, Dec. 2000, P1122 - 1146, Volume.26, Issue.12.

[NUN02] Nuno David, Jaime Simão Sichman, Helder Coelho, "Towards an Emergence-Driven Software Process for Agent-Based Simulation", Multi-Agent-Based Simulation II, Proceedings of MABS 2002, Third International Worshop

[OBJ02] Objective Engineering, Inc., "Modeling functional requirements with use cases: frequently asked questions",2002

[PER01] Pere Botella, Xavier Burgués, Xavier Franch, Mario Huerta, Guadalupe Salazar, Modeling Non-Functional Requirements, 2001

[PFL98] Pfleeger, S. L., Software Engineering - Theory and Practice. Prentice Hall, 1998.

[PRE97] Pressman, R. S., Software Engineering: A Practitioner's Approach (4th Edition). McGraw-Hill, Inc. 1997

[RAT00] Rational Unified Process, Version 2001, Rational Software, Cupertino, California, and Philippe Kruchten, The Rational Unified Process: An Introduction, 2e. Addison Wesley, 2000.

[RKA99] R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, S.G. Woods, "Expereience with Performing Archietcure Tradeoff Analysis"; ICSE 99, pp.54-63, 1999.

[ROM85] Rome Air Development Center (RADC). Software Quality Specification Guidebook RADC-TR-85-37, vol. II, 1985.

[RUT01] Ruth Malan and Dana Bredemeyer, "Defining Non-Functional Requirements", 2001, BREDEMEYER CONSULTING

[SEO99] Seo-Young Noh, Shashi K. Gadia, RAST: Requirement Analysis Support Tool based on Linguistic Information, 1999.

[SMI97] Smidts C., Sova D., Mandela G.K., An architectural model for software reliability quantification, PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, 2-5 Nov. 1997 P 324 – 335.

[SOW93] S. Owre, N. Shankar, and J. M. Rushby "User Guide for the PVS Specification and Verification System", Computer Science Lab, SRI International March 1993

[SOW95] S. Owre, J. Rushby, N. Shankar, and F. von Henke: "Formal verification for fault tolerant architectures Prolegomena to the design of PVS", IEEE Transactions on Software Engineering vol. 21 no. 2 pp. 107-125 February 1995

[STA97] The Standish Group's CHAOS Reports, Standish Group, 1994-1997

[TBO94] T. Boyce, "SafeFM case study report" Tech. Rep. SafeFM-018-GEC-1, SafeFM project, January 1994

[UML98] UML specification, OMG, 1998

[WEI93] Weinberg, Greland M., Quality Software Management, Vol2: First-Order Measurement, Dorset House, 1993.

[WIN70] Winston Royce, "Managing the Development of Large Software System." Proceedings of IEEE WESCON (August 1970), pp.1-9.

[WOL94] WolsongnNPP 2/3/4, "Software Work Practice Procedure for the Specification of SR for Safety Critical Systems," Design Document no. 00-68000-SWP-002, Rev. 0, Sept. 1991.ansaction on Software Engineering, vol.20, no.9, sept. 1994.

[XBU00] X. Burgués, X. Franch. "A Language for Stating Component Quality". Proceedings of 14th Brasilian Symposium on Software Engineering (SBES), Joao Pessoa (Brasil), October 2000, pp. 69-84.

[ZHI03] Zhiming Liu, He Jifeng, Xiaoshan Li and Yifeng Chen, A Relational Model for Formal Object-Oriented Requirement Analysis in UML, ICFEM 2003, 5-7 November, 2003, Singapore.