UNIVERSITA' DEGLI STUDI DI PISA

**DIPARTIMENTO DI INGEGNERIA DELLA INFORMAZIONE
ELETTRONICA, INFORMATICA, TELECOMUNICAZIONI**

TESI DI DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA, INFORMATICA E DELLE TELECOMUNICAZIONI

XVI CICLO

QUALITY EVALUATION OF NATURAL LANGUAGE SOFTWARE REQUIREMENTS: METHODS, TECHNIQUES AND TOOLS

Il candidato

Giuseppe Lami

I tutori

Prof. Stefania Gnesi **Prof. Luca Simoncini**

Acknowledgments

Thanks to my tutors Stefania Gnesi and Luca Simoncini for their continuous support and incitement.

The GUI of the QuARS tool has been designed in cooperation with Gianluca Trentanni. Gianluca took care of the QuARS GUI development.

The version of the QuARS tool described in this thesis has been realized in the framework of my Affiliation with the Software Engineering Institute of the Carnegie Mellon University of Pittsburgh, PA (U.S.A.). Several improvement hints have been gathered during my stays at the Software Engineering Institute. Special thanks for the encouragement in my effort for improving the QuARS tool to Mario R. Barbacci, Dennis Goldenson and Robert Ferguson of the Software Engineering Institute.

Thanks to Fabrizio Fabbrini, Alessandro Fantechi, Mario Fusani and Isabel John for the fruitful work made togheter in the last years.

Table of Contents

1. Introduction

The research activity I performed in the last tree years for my PhD. course belongs to the software engineering area and addresses some aspects of the software qualty evaluation. In this chapter some introductory, general considerations about software engineering and software quality are provided. The purpose and scope of this doctoral thesis are provided too.

1.1 Software Engineering

The Software Engineering term was invented in early ‘70s. In that time truly large software systems were attempted to be built commercially. The people on these large projects quikly realized that building large software systems was significantly different from building small systems. There were fundamental difficulties in scaling up the techniques of small-program development to large software development. Large software projects were universally over budget and behind schedule, this situation was called “software crisis”. Software Engineering was an attempt to apply engineering principles to the software development with the aim to provide software developers with a disciplinated development approach and repeatable practices for the whole software development. [90],[97]

Sommerville defines software engineering as [97]:

"an engineering discipline which is concerned with all aspects of software production from the early stages of system specifications through to maintaining the system after it has gone into use"

In this definition there are two key phrases:

"engineering discipline": engineers make things work. They apply theories, methods and tools where these are aprropriate but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods to support them. Engineers also recognize that they must work to organizational and financial constraints, so they look for solutions within the constraints.

"all aspects of software production": software engineering is not just concerned with the technical processes of software development but also with activities such as software projects management and with the development of tools, methods and theories to support software production.

Due to the peculiar nature of the software (software is essentiallly a design activity, the mere production of single instances of a product, differently from other engineering disciplines, is almost irrelevant), it is necessary a dual emphasis for facing the software challenges: the emphasis was put not only on the product (what is done) but also on the process (how things are done), because the awareness that high-quality processes should lead to high-quality products was achieved.

In order to make the development predictible, repeatable, measurable and efficient, the whole software development process was considered as composed of different single interdependent phases: it was called "software life-cycle". Different software life-cycles models have been defined, but they differ from each other mainly for the sequence and the number of times the main phases are perfomed. The main software development phases common to all the software life-cycle models are: Requirements Gathering & Analysis, Architectural Design, Detailed Design, Coding and Unit Testing, Software Integration, System Integration, Acceptance Testing.

In the following, some of the most popular life-cycle paradigms are described.

Water-fall life-cycle model: it was the first to be proposed in the early 70s and it is widely used even today. The basic idea is to perform the main steps of the software development in sequence. The advantages of this model are: the progress in the development are masureable, it makes possible to estimate the duration of the sigle steps on the basis of the experience applying steps in past projects, it allows the reuse of software artifact in other projects. The original water-fall model has also disadvantages because it disallowed iteration, it was inflexible, monolithic, it was difficult to perform estimations, and maintenance and it didn't behave well in case of requirements changing over time. For this reasons, the original water-fall model was modified in "water-fall with feedbacks" known also as the "V-Model". In figure 1.1 the V-Model life-cycle is schematically shown.

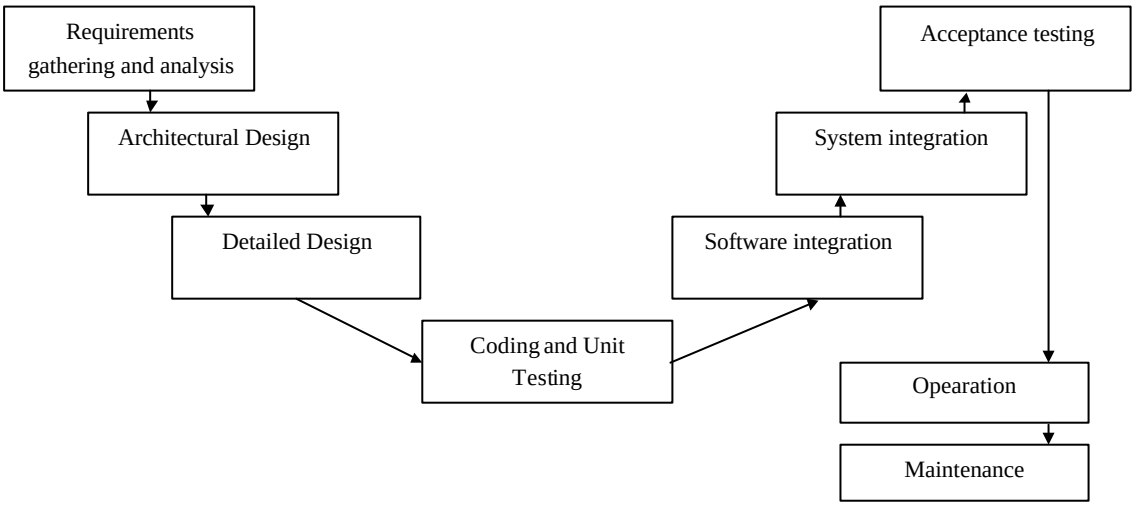


Figure 1.1: the V-Model software life-cycle

Rapid-prototyping model: in this approach some initial prototypes are build and used to develop requirements specification. Once the requirements are known the water-fall model is adopted. The prototypes are discarded once the design begins, they should not be used as a basis for implementation. Figure 1.2 shematically shows the rapid-prototyping model.

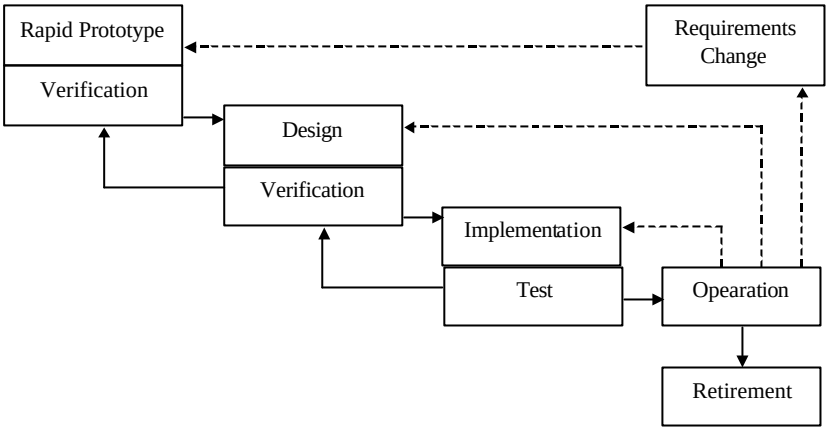


Figure 1.2: The Rapid Prototyping Model

Spiral life-cycle model: The spiral lifecycle model is the combination of the classic waterfall model and an element called *risk analysis.* This model is appropriate for large software projects. The model consists of four main parts, or blocks, and the process is shown by a continuous loop going from the outside towards the inside. This shows the progress of the project.

***Planning:*This phase is where the objectives, alternatives, and constraints are determined.**

Risk Analysis: **What happens here is that alternative solutions and constraints are defined, and risks are identified and analyzed. If risk analysis indicates uncertainty in the requirements, the prototyping model might be used to assist the situation.**

Engineering: **Here the customer decides when the next phase of planning and risk analysis occur. If it is determined that the risks are to high, the project can be terminated.**

Customer Evaluation: **In this phase, the customer will assess the engineering results and make changes if necessary.**

In Figure 1.3 the scheme of the Spiral model is provided.

The very new concept introduced by Software Engineering was the interdependence of all the phases of the software life-cycle and the equal relevance of each of them for the success of the the whole project.

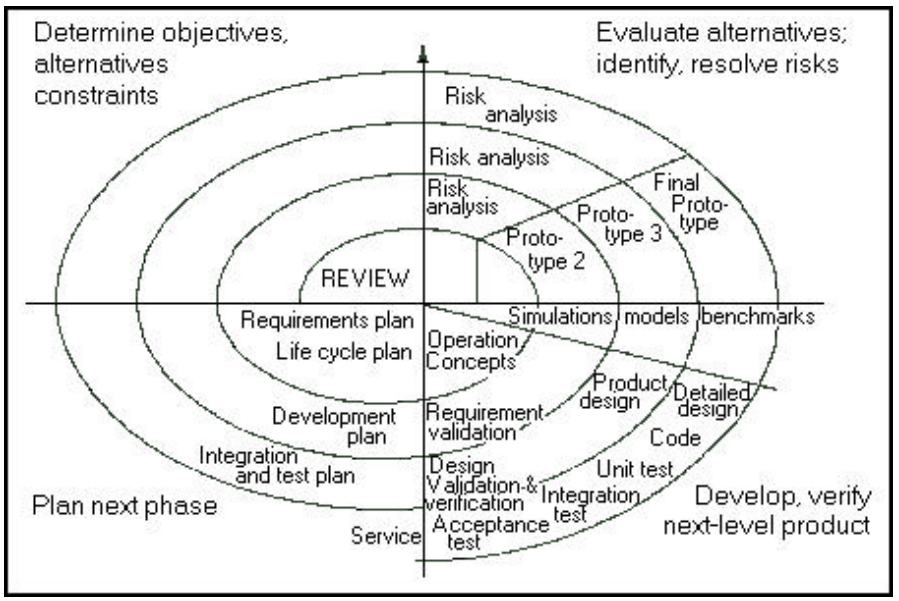


Figure 1.3: the Spiral life-cycle model

At the same time it became clear that the more the process was supported by tools and appropriate organizational infrastructures the more the improvements in the software field speeded-up. Software development environments were build, they are computer-based tools that are intended to assist the software development process. Development methods impose structure on the software activity with the goal of making the activity systematic and ultimately more likely to be successful.

1.2 Software quality

The more the software is spreaded in almost all the fields of human life, the more the concept of software quality became relevant [7]. In order to satify the needs of the users, being simply correct is not sufficient for software. Correctness is just one of properties the software has to fullfil to be suitable for its purpose. This consideration leaded to the concept of software quality.

The concept of quality referred to software is complex and variabile according to the different perspectives approached. It is possible to identify the following perspectives from which software quality can be seen:

Trascendental perspective: it derives from a perception of the quality of a software product as a whole that it isn't measurable but everyone can recognize when interacts with it. The trascendental quality can't be decomposed because it is an overall property. It is not possible to achieve a precise and quantitative evaluation of this kind of quality. [77]

User perspective: it can be defined as the extent a software product satisfies the needs and expecteations of an user in a operational context. This perception of the quality is necessarely based on what the software is required to do. Therefore, quality measurement has to be based on the availability and knowledge of the operational profiles (i.e. how the software product will be used during the usual operation).

Developer perspective: it can defined as the extent the software product satisfies the formalized requirements. In this case, the software quality can be measured (e.g. in terms of number of defects and cost for their correction).

Product perspective: the quality of the software derives from the inherent properties of the product itself. The overall quality, in this case, can be seen as the composition of several particular qualities. The measurement of the quality is made indirectly through the calculation of metrics that are supposed to measure (or estimate) the different qualities.

Value-based perspective: in this case the quality concept of a software product can be defined in terms of the trade-off between costs and benefits. This is the usual way the quality is evaluated by an acquirer.

For giving the software quality a common framework the standard ISO/IEC 9126 was defined. This standard provides a quality model for software, composed of quality characteristics, sub-characteristics and metrics. [59],[60],[61],[62]

Software product quality should be evaluated using a defined quality model. The quality model should be used when quality goals for software products and intermediate products are set. Software product quality should be hierarchically decomposed into a quality model composed of characteristics and subcharacteristics which can be used as a checklist of issues related to quality.

The whole quality of software has been, in the ISO/IEC 9126 standard, decomposed in tree levels:

***Internal quality*: is the totality of characteristics of the software product from an internal view. Internal quality is measured and evaluated against the internal quality requirements. Details of software product quality can be improved during code implementation, reviewing and testing, but the fundamental nature of the software product quality represented by internal quality remains unchanged unless redesigned.**

***External Quality* is the totality of characteristics of the software product from an external view. It is the quality when the software is executed, which is typically measured and evaluated while testing in a simulated environment with simulated data using external metrics. During testing, most faults should be discovered and eliminated. However, some faults may still remain after testing. As it is difficult to correct the software architecture or other fundamental design aspects of the software, the fundamental design usually remains unchanged throughout testing.**

***Quality in Use* is the user's view of the quality of the software product when it is used in a specific environment and a specific context of use. It measures the extent to which users can achieve their goals in a particular environment, rather than measuring the properties of the software itself**

The metrics are the means to evaluate the quality characteristics of a software product in a quantitative way [42]. To measure internal, external quality and quality in use of a software product different metrics are required. The ISO/IEC 9126 standard defines tree cathegories of metrics:

***Internal metrics*: can be applied to a non-executable software product (such as a specification or source code) during designing and coding. When developing a software product the intermediate products should be evaluated using internal metrics which measure intrinsic properties, including those which can be derived from simulated behaviour. The primary purpose of these internal metrics is to ensure that the required external quality and quality in use is achieved. Internal metrics provide users, evaluators, testers, and developers with the benefit that they are able to evaluate software product quality and address quality issues early before the software product becomes executable.**

***External metrics*: use measures of a software product derived from measures of the behaviour of the system of which it is a part, by testing, operating and observing the executable software or system. Before acquiring or using a software product it should be evaluated using metrics based on business objectives related to the use, exploitation and management of the product in a specified organisational and technical environment. External metrics provide users, evaluators, testers, and developers with the benefit that they are able to evaluate software product quality during testing or operation.**

***Quality in use metrics*: measure the extent to which a product meets the needs of specified users to achieve specified goals with effectiveness, productivity, safety and satisfaction in a specified context of use . Evaluating quality in use validates software product quality in specific user-task scenarios. Quality in use is the user's view of the quality of a system containing software, and is measured in terms of the result of using the software, rather than properties of the software itself. Quality in use is the combined effect of internal and external quality for the user.**

Software product quality should be evaluated using a defined quality model. The quality model should be used for setting quality goals for software products and intermediate products. Software product quality should be hierarchically decomposed into a quality model composed of characteristics and subcharacteristics which can be used as a checklist of issues related to quality [70].

It is not practically possible to measure all internal and external subcharacteristics for all parts of a large software product [9]. Similarly it is not usually practical to measure quality in use for all possible user-task scenarios. Resources for evaluation need to be allocated between the different types of measurement dependent on the business objectives and the nature of the product and design processes.

In figure 1.4 the ISO/IEC 9126 quality model for internal and external quality is graphically shown. For more details and the definitions of the quality characteristics and sub-characteristics, please refer to the standard [59]. This quality model categorises software quality attributes into six characteristics (functionality, reliability, usability, efficiency, maintainability and portability), which are further subdivided into subcharacteristics. The subcharacteristics can be measured by internal or external metrics [52].

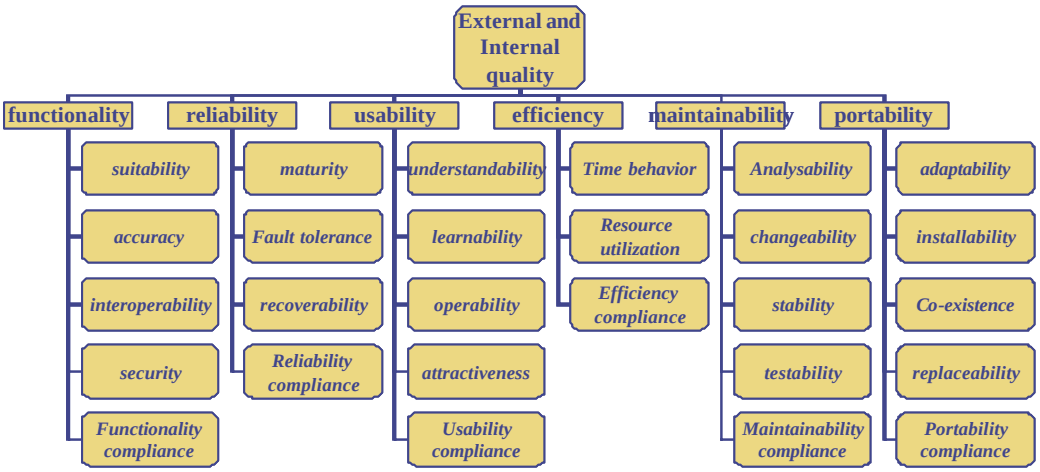


Figure 1.4. Qualty model for Internal and External software quality

Figure 1.5 shows the quality model for quality in use. The attributes of quality in use are categorised into four characteristics: effectiveness, productivity, safety and satisfaction (Figure 1.5). Quality in use is the user’s view of quality. Achieving quality in use is dependent on achieving the necessary external quality, which, in turn, is dependent on achieving the necessary internal quality

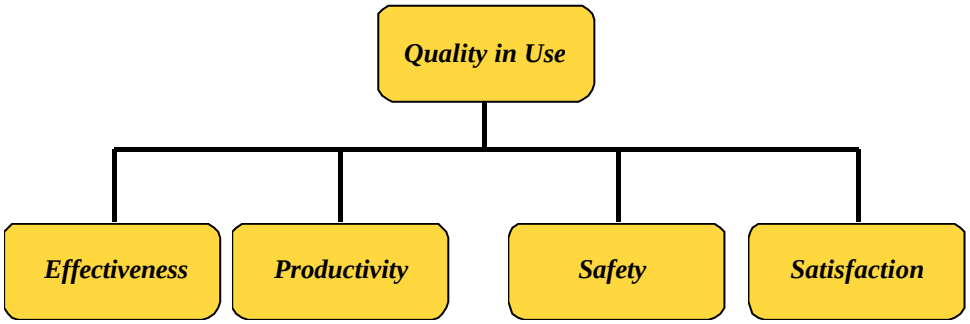


Figure 1.5: The Quality in Use quality model

The ISO/IEC 9126 standard is generally recognized as the principal reference for the quality of software products. The view of the quality of software products as composed of (sub-)characterisitcs to be measured by means of metrics contained in the ISO/IEC 9126 standard is the same view of quality I took during my work.

1.3 Purpose and Scope of this Thesis

Software is “all or part of the programs, procedures, rules, and associated documentation of an information processing system” [ISO/IEC 2382-1: 1993] [56]. The quality of software is composed, as said in Section 1.2, of several characteristics and can’t be faced as a whole.

The purpose of this thesis is to contribute to the definition of techniques and methods to evaluate and improve the quality of the software and to the provision of automatic tools that can make them practically usable. I concentrated my research activity on one of the components of a software product scarcely supported by methods and automatic tools: the requirements documents written in natural language (NL). I started my research work with an investigation of the demands and needs in the requirements area and then I used the outcomes of that investigation to target the rest of my work.

The scope of this PhD thesis is composed of the following points:

the definition of a quality model for NL requirement

the identification of feasible and effective techniques for performing NL requirements analysis aiming at pointing out defects

the implementation of a tool to automatize this analysis

the investigation of the applicability of the defined approach in the practice

In the following chapters these points are treated and the achieved outcomes are described and discussed.

2. Requirements Engineering

In this chapter some introductory considerations about what the Requirements Engineering process is, are provided. The outcomes of this process are treated, playing particular attention to one of them: the *requirements specification document* that is the object of the research activity described in the next chapters.

2.1 Requirements Engineering

The Requirements Engineering for software is concerned with the acquisition, analysis, specification, validation and management of software requirements. It is widely acknowledged within the software industry that software projects are critically vulnerable when these activities are performed poorly. This has led to the widespread use of the term ‘requirements engineering’ to denote the systematic handling of requirements. Software requirements are one of the products of the requirements engineering process.

Software requirements express the needs and constraints that are placed upon a software product that contribute to the satisfaction of some real world application. The application may be, for example, to solve some business problem or exploit a business opportunity offered by a new market. It is important to understand that,

except where the problem is motivated by technology, the problem is an artifact of the problem domain and is generally technology neutral. The software product alone may satisfy this need (for example, if it is a desktop application), or it may be a component (for example, a speech compression module used in a mobile phone) of a software-intensive system for which the satisfaction of the need is an emergent property. In fundamental terms, the way in which the requirements are handled for stand-alone products and components of software-intensive systems is the same.

One of the main objectives of requirements engineering is to discover how to partition the system; to identify which requirements should be allocated to which components. In some systems, all the components will be implemented in software. Others will comprise a mixture of technologies. Almost all will have human users and sometimes it makes sense to consider all components of the system to which requirements should be allocated (for example, to save costs or to exploit human adaptability and resource fulness).

Because of this requirements engineering is fundamentally an activity of systems engineering rather than one that is specific to software engineering. In this respect, the term 'software requirements engineering' is misleading because it implies a narrow scope concerned only with the handling of requirements that have already been acquired and allocated to software components.

One of the fundamental tenets of good software engineering is that there is good communication between system users and system developers. It is the requirements engineer who is the conduit for this communication. They must mediate between the domain of the system user (and other stakeholders) and the technical world of the software engineer. This requires that they possess technical skills, an ability to acquire an understanding of the application domain, and the inter-personal skills to help build consensus between heterogeneous groups of stakeholders [87].

The process of requirements engineering is composed of four main phases: [17], [97]

Requirements Elicitation: in this phase the software requirements are elicited by taking information from the potential stakeholders, by analysis the domain and by considering the existing standards of interests;

Requirements analysis and negotiation: in this phase the elicited requirements, that can be still expressed in a non structured way, are analysed to identify faults, inconsistencies, incompleteness. For doing that several constraints are taken into account (businees, technical, schedule, regulatory and other constraints). After a negotiation among the different parties involved in the software project an agree version of the requirements is achieved.

Requirements Specification: the agree requirements are moved in a document meeting the required structure, quality and verifiability.

Requirements validation: the requirements docuement is validated

If, after the validation, the document is not accepted the four steps are repeated (see figure 2.1).

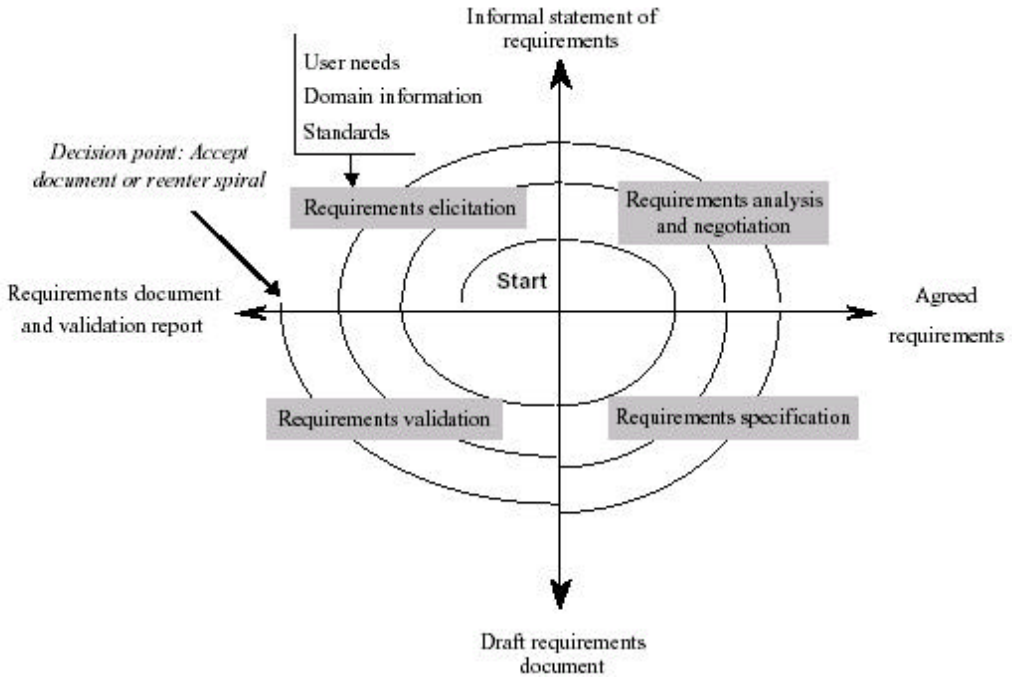


Figure 2.1: The requirements engineering process

2.2 What is a Requirement

At its most basic, a requirement is a property that must be exhibited in order to solve some problem of the real world. Hence, a requirement is a property that must be exhibited by a system developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the system, to support the business processes of the organisation that has commissioned the system, to correct shortcomings of an existing system, to control a device and many more. The functioning of users, business processes and devices are typically complex. By extension, therefore, the requirements on a system are typically a comple combination of requirements from different people at different levels of an organisation and from the environment in which the system must operate [49].

Requirements vary in intent and in the kinds of properties they represent. A distinction can be drawn between *product parameters* and *process parameters*. Product parameters are requirements on the system to be developed and can be further classified as:

? ??**Functional requirements on the system such as formatting some text or modulating a signal. Functional requirements are sometimes known as capabilities.**

? ??**Non-functional requirements that act to constrain the solution. Non-functional requirements are sometimes known as constraints or quality requirements. They can be further classified according to whether they are (for example) performance requirements, maintainability requirements, safety requirements, reliability requirements, electro-magnetic compatibility requirements and many other types of requirements.**

A process parameter is essentially a constraint on the development of the system (e.g. ‘the software shall be written in Ada’). These are sometimes known as process requirements.

Requirements must be stated clearly and unambiguously and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements that depend for their interpretation on subjective judgement (‘the system shall be reliable’, ‘the system shall be user-friendly’).

Two examples of quantified requirements are: that a system must increase a call-center’s throughput by 20%; and a requirement that a system shall have a probability

of generating a fatal error during any hour of operation of less than 1 * 10^{-8}. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture [16, 97].

An essential property of all requirements is that they should be verifiable. It may be difficult or costly to verify certain requirements. For example, verification of the throughput requirement on the call-center may necessitate the development of simulation software. The requirements engineering and V&V personnel must ensure that the requirements can be verified within the available resource constraints.

Some requirements generate implicit process requirements. The choice of verification method is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce systemic errors that can lead to inadequate reliability.

Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator. Requirements have other attributes in addition to the behavioural property that they express. Common examples include a priority rating to enable trade-offs in the face of finite resources and a status value to enable project progress to me monitored. Every requirement must be uniquely identified so that they can be subjected to configuration control and managed over the entire system life cycle.

2.3 The Requirements Document

Good requirements engineering requires that the products of the process - the deliverables - are defined [90], [99]. The most fundamental of these in requirements engineering is the requirements document. This often comprises two separate documents:

A document that specifies the system requirements. **This is sometimes known as the requirements definition document, user requirements document or, as defined by**

IEEE std 1362-1998 [55], the concept of operations (ConOps) document. This document serves to define the high-level system requirements from the stakeholders' perspective(s). It also serves as a vehicle for validating the system requirements. Its readership includes representatives of the system stakeholders. It must therefore be couched in terms of the customer's domain. In addition to a list of the system requirements, the requirements definition needs to include background information such as statements of the overall objectives for the system, a description of its target environment and a statement of the constraints and nonfunctional requirements on the system. It may include conceptual models designed to illustrate the system context, usage scenarios, the principal domain entities, and data, information and work flows [103].

A document that specifies the software requirements. **This is sometimes known as the software requirements specification (SRS). The purpose and readership of the SRS is somewhat different than the requirements definition document. In crude terms, the SRS documents the detailed requirements derived from the system requirements, and which have been allocated to software. The non-functional requirements in the requirements definition should have been elaborated and quantified. The principal readership of the SRS can be assumed to have some knowledge of software engineering concepts. This can be reflected in the language and notations used to describe the requirements, and in the detail of models used to illustrate the system. For custom software, the SRS may form the basis of a contract between the developer and customer [72, 102].**

Requirements documents must be structured so as to minimize the effort needed to read and locate information within them [10], [86], [99]. Failure to achieve this reduces the likelihood that the system will conform to the requirements. It also hinders the ability to make controlled changes to the document as the system and its requirements evolve over time. Standards such as IEEE std 1362-1998 [55] and IEEE std 830-1998 [54] provide templates for requirements documents. Such standards are intended to be generic and need to be tailored to the context in which they are used.

Care must also be taken to describe requirements as precisely as possible. Requirements are usually written in natural language but in the SRS this may be supplemented by formal or semi-formal descriptions. Selection of appropriate notations permits particular requirements and aspects of the system architecture to be described more precisely and concisely than natural language. The general rule is that

notations should be used that allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical and certain other types of dependable systems. However, the choice of notation is often constrained by the training, skills and preferences of the document's authors and readers.

Natural language has many serious shortcomings as a medium for description. Among the most serious are that it is ambiguous and hard to describe complex concepts precisely. Formal notations such as Z or CSP [100], [101] avoid the ambiguity problem because their syntax and semantics are formally defined. However, such notations are not expressive enough to adequately describe every system aspect. Natural language, by contrast, is extraordinarily rich and able to describe, however imperfectly, almost any concept or system property. A natural language is also likely to be the document author and readerships' only *lingua franca*. Because natural language is unavoidable, requirements engineers must be trained to use language simply, concisely and to avoid common causes of mistaken interpretation. These include:

? ?**long sentences with complex sub-clauses;**

? ?**the use of terms with more than one plausible interpretation (ambiguity);**

? ?**presenting several requirements as a single requirement;**

? ?**inconsistency in the use of terms such as the use of synonyms.**

To counteract these problems, requirements descriptions often adopt a stylized form and use a restricted subset of a natural language. It is good practice, for example, to standardize on a small set of modal verbs to indicate relative priorities. For example, 'shall' is commonly used to indicate that a require ment is mandatory, and 'should' to indicate a requirement that is merely desirable. Hence, the requirement 'The emergency breaks shall be applied to bring the train to a stop if the nose of the train passes a signal at DANGER' is mandatory.

The requirements documents(s) must be subject to validation and verification procedures. The requirements must be validated to ensure that the requirements engineer has understood the requirements. It is also important to verify that a

requirements document conforms to company standards, and is understandable, consistent and complete. Formal notations offer the important advantage that they permit the last two properties to be proven (in a restricted sense, at least). The document(s) should be subjected to review by different stakeholders including representatives of the customer and developer. Crucially, requirements documents must be placed under the same configuration management regime as the other deliverables of the development process [10], [95].

The requirements document(s) are only the most visible manifestation of the requirements. They exclude information that is not required by the document readership. However this other information is needed in order to manage them. In particular, it is essential tha requirements are traced.

One method for tracing requirements is through the construction of a directed acyclic graph (DAG) that records the derivation of requirements and provides audit trails of requirements. As a minimum, requirements need to be traceable backwards to their source (e.g. from a software requirement back to the system requirement(s) from which it was elaborated), and forwards to the design or implementation artifacts that implement them (e.g. from a software requirement to the design document for a component that implements it). Tracing allows the requirements to be managed. In particular, it allows an impact analysis to be performed for a proposed change to one of the requirements.

Modern requirements management tools help maintain tracing information [50]. They typically comprise a database of requirements and a graphical user interface:

? ??**to store the requirement descriptions and attributes;**

? ??**to allow the trace DAGs to be generated automatically;**

? ??**to allow the propagation of requirements changes to be depicted graphically;**

? ??**to generate reports on the status of requirements (such as whether they have been analysed, approved, implemented, etc.);**

? ??**to generate requirements documents that conform to selected standards;**

? ?**and to apply configuration management to the requirements.**

It should be noted that not every organisation has a culture of documenting and managing requirements. It is common for dynamic start-up companies which are driven by a strong ‘product vision’ and limited resources to view requirements documentation as an unnecessary overhead [88], [96]. Inevitably, however, as these companies expand, as their customer base grows and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and management are fundamental to the any requirements engineering process.

3. Research Directions: the ESCAPE Project

In this chapter the outcomes of an experience in software process assessment I held in the framework of the ESCAPE project are described. This experience has been conducted with the aim to evaluating the capability of the software development process of the FIAT Auto's software suppliers. The outcomes, of this project, in particular those related to the Requirements Elicitation and Analysis processes, have been useful for understanding the industry's demands and needs in the field of requirements engineering. My participation in the ESCAPE project has been a unvaluable opportunity to achieve a deep and direct knowledge of the software development process of many highly mature companies Europe-wide and to concentrate my research effort towards those areas resulting particularly critical in terms of source of errors and troubles for software projects. In the following the way the ESCAPE project has been conducted and the outcomes of interest are described.

3.1 Automotive software

The past four decades have witnessed an exponential increase in the number and sophistication of electronic systems in vehicles. The growth of electronic systems has had implications for vehicle engineering and the resulting demands on power and design have led to innovations in electronic networks for automobiles [75]. Just as

LANs connect computers, control area networks (CANs) connect a vehicle's electronic equipment and facilitate the sharing of information and resources among the distributed applications. A typical vehicle can contain several CANs, operating at different transmission rates, to manage a car's "comfort electronics" (like seat and window movement), or to run more real-time critical functions (like cruise control, antilock brakes and engine management). Other applications that use electronics to control a system (the so-called X-by-wire solutions), rather than mechanical or hydraulic means, are responsible for an ongoing revolution in vehicle electronics architecture and require more specialized and reliable control networks. Multimedia devices in automobiles and interconnecting facilities over the Internet demand networks with extensive bandwidth, while other applications require wireless configurations [71]. Vehicles are becoming more like PCs, allowing for a number of plug-and-play devices and creating the potential for significant growth in automotive application software [104]. With more than 85% of the functionality in the modern motor vehicle already controlled by software, both the motor vehicle manufacturer and the software supplier need to take action to face quality issues related to the management of software projects.

Suppliers have not been slow to take action. A number of companies are already using process assessment techniques as a basis to identify areas for improvement in their processes both to meet their business needs and the demands of their customers. There is a general understanding that action has to be taken if they are to continue to make business.

The motor vehicle manufacturers, as a general trend, have also started to take pro-active action to address the situation in a number of ways; by focusing on software capability of the supplier in the supplier evaluation process; making provision for contractual demands with respect to software quality; performing supplier software capability assessments both before and during contract performance; and asking for the supplier to implement process improvement plans, when needed.

To set up a methodology supporting the management of software projects and suppliers, Fiat Auto started the ESCAPE (Electronics Software Capability Evaluation) Project, in co-operation with the System and Software Evaluation Centre (SSEC - an independent organism of the Italian National Research Council that performs evaluation and certification activity in Information Technology) [20],[26], [28], [29], [30], [31], [32],[33], [34], [35] with the following main goals:

To improve the software suppliers selection process.

To provide Fiat Auto with methods to determine the risks associated to a software supplier.

To improve the software development process of suppliers, helping them to detect possible weaknesses and risks in specific processes, to define improvement paths and to provide tools for verifying the results of improvement actions

To achieve a better control on the software development project and on the quality of the resulting product.

3.2 The ESCAPE Project

The need to evaluate the Process Capability of the Suppliers of FIAT Auto was discussed between the SSEC and FIAT Auto in year 2000. The main goals and reasons for undertaking this evaluation were:

To derive a "capability" and "risk" level for each software supplier.

To improve the supplier selection process of FIAT Auto by using criteria based on the derived supplier "capability" and "risks".

To improve the control of the supplier's software development process and of the quality of the resulting products.

To identify weaknesses and strengths of the supplier's software process.

To identify possible improvement opportunities in the relations between the customer and suppliers.

With the aim to achieve the above targets, the ESCAPE project was started by FIAT and, in the framework of this project, cooperation with SSEC was established.

The first step of the ESCAPE project was to decide the reference model to use to perform the suppliers evaluation. The traditional reliance on Quality Systems Standards such as ISO9001 [57] and QS9000 [91] has not provided sufficient confidence in the software area. The motor vehicle manufacturers, like others in the defence and aerospace industries, have now turned to international standards for software process assessment, based on ISO 15504 [59] and/or the Capability Maturity Model (CMM), as a means to identify and control risk and to assess the software capability of suppliers [85], [47], [68]. Some claimed features of the ISO 15504 standard proved to be crucial in selecting the approach to perform the software capability assessments [19]:

Software-oriented approach

Applicability over a wide range of application domains, businesses and sizes of organizations

Output as process profiles at different levels of detail

Comparability, reliability and consistency of results

Independence of organizational structures, life cycle models, technologies and development models

Adaptability of the assessment scope to cover specific processes of interest

Re-usability of assessment results, both for process improvement and capability determination

Another important factor that supported the decision to use SPICE as the assessment methodology is the launch of an initiative by the Procurement Forum (www.procurementforum.org) with the principal European Car Makers, their assessors and representative bodies to address the problems related to software assessments in automotive. In the framework of this initiative, a Special Interest Group has been founded with the aim to design a special version of the SPICE model (called Auto-SPICE) tailored on the needs and peculiarities of the automotive business area. In fact, the focus on software capability determination by means of software

process assessment has already in use provided significant business benefits, but at the same time has highlighted the scale of the potential problem, particularly with suppliers of safety-critical embedded software system components. Whilst the immediate short-term benefits are clear for the motor vehicle manufacturer, in the near term, without consensus on commonality of approach, suppliers face multiple assessments from multiple manufacturers using different model and consuming resources that put additional pressure on delivery times. Therefore the choice of SPICE as the Reference Model to adopt for the Supplier's software process assessment has been corroborated by the existence of this European trend towards the use of the SPICE Model is in act in automotive.

FIAT Auto and the SSEC are part of this Special Interest Group and are currently participating in the works.

3.2.1 Assessment purpose and scope

The assessment purpose is to evaluate the software process capability of suppliers in order to improve the general project management during the development phases. In order to achieve a trade-off between performing a wide and comprehensive assessment that should provide many indications on the way a supplier conducts its own software development process and the need to respect budget limitations, FIAT Auto identified some critical areas concerning software with the aim to concentrate the assessment effort on them. The principal critical areas that was been identified are listed below:

Relationships between the customer (FIAT Auto) and suppliers

Extent of Requirement Analysis (by the supplier's side)

System design capability (where the system is intended as the ECU - Electronic Control Unit)

Software design capability

System integration and testing capability

Project management capability

Consequently, the scope of the assessment has been defined matching the criticalities identified by FIAT Auto with the processes in the SPICE reference model. The final scope of the assessments is composed of five processes that in the following are indicated according to the SPICE terminology:

CUS 3: Requirements Elicitation Process

ENG 1.1: System Requirements Analysis and Design Process

ENG 1.3: Software Design Process

ENG 1.7: System Integration and Testing Process

MAN 2: Project Management Process

3.3 Assessment Activities

The first step in the ESCAPE project was the selection of the suppliers to involve in the assessments and in each assessment the selection of the project (or process instances) to be considered in order to collect evidences of the process capabilities. The general policy followed gave priority to including those companies that are currently involved in new projects with FIAT Auto, and those projects with FIAT Auto that are enough advanced to provide sufficient evidence at assessment time.

The activities strictly related to the assessment were divided into four main phases:

***Preliminary meeting*: an introductory meeting was held at Fiat Auto at the beginning of the operational phase with representatives from the companies involved in the assessments, with the purpose of presenting the SPICE approach, of reviewing the assessment purpose, agreeing the scope and discussing the constraints, introducing the assessment activities and a provisional assessment plan. At this stage, particular care was taken on informing the suppliers that process assessment does not disclose**

sensitive information about the techniques used in software development nor details on proprietary software and algorithms. In fact, the assessment method intends investigate only on knowledge, experience, skill, confidence, benefits, resources allocation and management.

***Assessment preparation*: each assessee was sent a questionnaire - to fill and return before the on-site visit - to gather preliminary information on the processes. Furthermore, some documents describing the purpose and the topics to be investigated and the way the assessment should be conducted was sent too, to help the assessee to prepare for the assessment. These preliminary activities allow to save time during the on–site visit and to make of the assessment more effective and efficient. At this stage, a non-disclosure agreement was signed by the assessment team members.**

***On-site activities*: during the on-site visit (that was taken about 3-4 working days), and initial briefing was held aiming at recalling the assessment purpose, scope, constraints and model. Then, information was gathered by means of presentations, document analysis and interviews. To better assure assessment repeatability and results comparability, checklists were developed to be used as guidelines for the assessors.**

***Results derivation*: after the on-site visit the gathered data was validated and analysed and, for each assessed process, each process attribute was rated. Then, these ratings were used to derive the capability profiles and the capability levels of the assessed processes. Finally, a detailed report of the whole assessment was prepared including the detailed ratings and the final capability profiles. These results, along with the indication of improvement opportunities were sent to the company and to the sponsor (FIAT Auto).**

3.4 Outcomes

In this section the results in terms of Capability Profiles of the processes assessed are presented. For confidentiality reasons the name of the Companies where the assessments have been conducted has been omitted and the projects have been indicated as Pi (with i = 1, … , 10).

The figure 3.1 shows the detailed capability profiles of all the processes assessed. The capability pro file is the collection of the rating achieved by each process attribute of the Spice Model. According to the SPICE terminology and rules, an attribute is rated in a four-value scale (N=Not Achieved, P = Partially Achieved, L = Largely Achieved, F = Fully achieved). Furthermore, the bold red line determines the achieved capability level of the correspondent process.

Each grey column means that the correspondent process was, for some reason, excluded from the scope for that project.

		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	Capabilty Level
CUS 3	PA 4.2	N	N	N	N	N		N	N	N	N	4
	PA 4.1	N	N	N	N	N		N	N	N	N	
	PA 3.2	P	P	P	P	L		L	P	L	L	3
	PA 3.1	P	P	L	P	P		P	P	P	P	
	PA 2.2	L	L	P	P	L		F	L	F	L	2
	PA 2.1	L	L	L	L	L		F	F	F	L	
	PA 1.1	F	F	L	F	F		F	F	F	F	1
ENG1.1	PA 4.2	N	N	P	N	N	L		N	N	N	4
	PA 4.1	N	N	P	N	P	P		N	N	N	
	PA 3.2	L	L	L	L	L	L		P	L	L	3
	PA 3.1	P	P	L	P	P	L		N	P	P	
	PA 2.2	F	F	L	L	L	F		P	F	F	2
	PA 2.1	L	L	L	F	F	F		F	F	F	
	PA 1.1	F	F	F	F	F	F		L	F	F	1
ENG1.3	PA 4.2	N	N	P	N	P	L	L	N		N	4
	PA 4.1	N	N	L	N	P	P	N	P		N	
	PA 3.2	L	L	L	L	L	F	L	F		L	3
	PA 3.1	P	P	L	L	L	L	L	L		P	
	PA 2.2	L	F	F	F	L	F	F	F		F	2
	PA 2.1	L	L	F	F	F	F	F	F		F	
	PA 1.1	F	F	F	F	F	F	F	F		F	1
ENG1.7	PA 4.2	N	N	P	N	P	L	N	N		N	4
	PA 4.1	P	P	P	N	P	P	P	P		N	
	PA 3.2	F	L	L	L	L	F	L	L		L	3
	PA 3.1	L	P	L	P	L	F	P	P		P	
	PA 2.2	L	L	L	L	L	F	F	F		F	2
	PA 2.1	L	L	F	L	F	F	F	F		L	
	PA 1.1	F	F	F	F	F	F	F	F		F	1
MAN2	PA 4.2	N	N	N	N	N		L	N	N	N	4
	PA 4.1	N	N	N	N	N		P	P	N	P	
	PA 3.2	P	P	P	P	P		L	L	P	L	3
	PA 3.1	P	P	P	L	P		F	L	L	L	
	PA 2.2	P	P	P	P	L		L	F	F	F	2
	PA 2.1	P	P	L	L	P		F	L	F	F	
	PA 1.1	L	L	F	F	F		F	F	F	F	1

Figure 3.1. The complete capability profile of the process assessed

From the data above some general indications may be derived on the process capabilities of the automotive software suppliers belonging to the sample considered. In fact, as the Figure 3.2 shows, the average ratings of the assessed processes provide some high level indications.

The average capability level achieved by the technical processes (ENG.1.1, ENG.1.3 and ENG.1.7) is higher than for the other processes. From this outcome it is possible to infer that the weaker areas are not related to the capability to perform the technical tasks (see the capability level of the engineering processes) but are related mainly to the managerial issues and the relations with the customer.

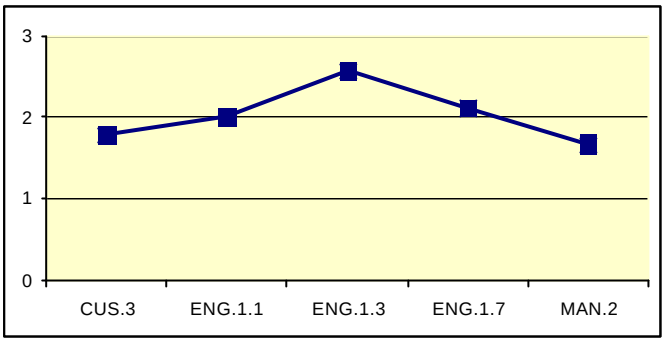


Figure 3.2. Average ratings

3.4.1 General Considerations

From the assessments results, some general issues emerge. The principal issues will be discussed below together with their consequences and the related improvement opportunities. These issues identify also research directions to provide answers to real needs of the automotive industry

3.4.2 Requirements Engineering in Practice

The overview of the requirements engineering process given in section 2.1 described it as if it was a linear sequence of activities. This is an idealised view of the process. In this section some reasons, emerged from the outcomes of the ESCAPE project, why a linear process is seldom practicable in the context of real software projects are examinated.

There is a general pressure in the software industry for ever-shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, most projects are constrained in some way by their environment and many are upgrades to or revisions of existing systems where the system architecture is a given. In practice, therefore, it is almost always impractical to implement requirements engineering as a linear, deterministic process where system requirements are elicited from the stakeholders, baselined, allocated and handed over

to the software development team. It is certainly a myth that the requirements for large systems are ever perfectly understood or perfectly specified [97].

Instead, requirements typically iterate toward a level of quality and detail that is sufficient to permit design and procurement decisions to me made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the development process. However, requirements engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the requirements' quality is as high as possible given the available resources. They should, for example, make explicit any assumptions that underpin the requirements, and any known problems. Even where requirements engineering is well resourced, the level of analysis will seldom be uniformly applied. For example, early in the analysis process experienced engineers are often able to identify where existing or off-the-shelf solutions can be adapted to the implementation of system components. The requirements allocated to these need not be elaborated further, while others, for which a solution is less obvious, may need to be subjected to further analysis. Critical requirements, must always be analyzed rigorously. In almost all cases requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point of understanding about requirements engineering is that a significant proportion of the requirements *will* change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the 'environment': the customer's operating or business environment; or in the market into which the system must sell, for example.

Whatever the cause, it is important to recognise the inevitability of change and adopt measures to mitigate the effects of change. Change has to be managed by ensuring that proposed changes go through a defined review and approval process, and by applying careful requirements tracing, impact analysis and version management. Hence, the requirements engineering process is not merely a frontend task to software development, but spans the whole development life cycle. In a typical project the activities of the requirements engineer evolve over time from elicitation to change management.

Another practice common to all the companies we interact in the framework of the ESCAPE project is the use of natural language for expressing requirements. In particular, the generally used language is English.

3.4.3 Other outcomes

In this section other outcomes emerged from the analysis of the outcomes of the ESCAPE project are described. They are mainly waeknesses found in the software processes of the companies assessed, these weaknesses are described along the consequences and identified improvement opportunities.

3.4.3.1 Facing the New Challenges in the Automotive Software

A few years ago the automotive development environment was still almost totally oriented to the system (intended as mechanics and electro-technical issues). In the last few years electronics and software have pervaded the automobiles and the automotive companies have to face this new challenge. The answer to the demand for an extensive use of electronic and software solutions is still inadequate, as it comes from an environment (including both customers and suppliers) that is not enough prepared for the transition - for historical, cultural and technical reasons.

Consequences:

The companies (at the top management level) are aware of these problems and are introducing (or have already introduced) new development process models with considerably high costs.

Several problems usually arise (consistently with the low capability level achieved by the CUS.3 process) about the mutual and unique understanding and agreement of the requirements.

The customer, which has the most difficult task (since it has to maintain a comprehensive point of view on the whole automobile product), is not completely aware of its role in the acquisition process and of the importance of its co-operation in it.

A similar situation may be found also within the supplier's organization between the departments dealing with the system and those dealing only with software components. This is most true in those companies that produce "traditional" components with embedded software.

Improvement opportunities:

To define reliable, available and competent interfaces between customer and supplier in order to facilitate their relations.

To increase the effort for identifying and managing the risks related to interpretation problems and poor involvement of one of the partners in the acquisition process.

To encourage initiatives among carmakers and software suppliers aimed at the definition of common schemes and standards (as for example Auto-SPICE).

3.4.3.2 Platforms vs. Ad Hoc Solutions

Automotive software is embedded in subsystems delivered in very large numbers (by the millions), with hard constraints in terms of resource optimization (e.g. memory size, computing power). Furthermore, the automotive software suppliers deliver their products to almost all the European carmakers, but they have to differentiate quality and performance characteristics of the products, to meet quality and cost requirements of the different carmakers.

Consequences:

The supplier tries to impose its own platforms, i.e. products based on the same architecture but with adaptable components. This kind of solution allows the supplier to save many resources.

The customer tries to get ad hoc designed software with the aim to maintain the full ownership of both the project and the marketing policies and reduce the dependence from a particular supplier.

Code minimization techniques have been taken up again in order to save memory occupation.

Improvement opportunities:

Both the customer and the supplier are scarcely aware of the risks related to the solutions adopted to cope with this problem (both the ad hoc and platform solutions may present significant drawbacks in the maintenance and upgrading phase). These risks have to be clearly understood and a trade-off between efficiency and resource utilization has to be achieved and managed.

3.4.3.3 The Modularity-related Issues

The electronic final system is characterized by a strong modularity since it is composed of integrated subsystems (typically ECUs)

Consequences:

Serious interoperability problems may arise (typically a stand-alone ECU runs correctly but behaves unpredictably when integrated with the others).

Improvement opportunities:

To adopt techniques for the requirements and interfaces specification and analysis in order to be able to better evaluate and manage their completeness.

3.5 Research Directions

From the outcomes of the ESCAPE project some research opportunities for bridging some gaps pointed out with the ESCAPE project have been derived.

In particular, some clear indications arose about the requirements analysis. Requirements are a very important work product because they represent the principal communication channel between the customer and the supplier. It is arisen

that the analysis of requirements is a crucial step both at the beginning of a project and during the development and also later. At the beginning of the development it is necessary evaluate the requirements in order to point out and possibly solve ambiguities and inconsistency in it for starting the project well. Because the volatility of customer requirements (they usually change often during the project) it is important to have the capability to continue their analysis. Testing (mainly the acceptence testing) is driven by the requirement then the testability of requirement has to be evaluated too.

Unfortunately, the outcomes of the ESCAPE project enlighted a lack of a systematic approach to the analysis of requirements, along with a lack of supportiing automatic tool (the requirements analysis, when made, is based on human reviews). This situation encouraged the reasearch activity I was performing.

4. The Quality of NL Requirements

In this chapter the quality characteristics the Natural Language (NL) requirements are have to fulfill are discussed. Then some considerations about the different approaches that can be adopted to evaluate the quality of Natural Language requirements are presented. Among these approaches the one based on the application of linguistic techniques has been privileged and a Quality Model for Natural Language requirements, composed of a set of quality charateristics and the related metrics, has been defined. The Quality Model presented in this chapter is the basis of the automatic tool for Natural Language (NL) requirements analysis presented in chapter 5. Finally, the way consistency and completeness characteristics of the NL requirements can be addressed by means of linguistic techniques is discussed.

4.1 Quality Characterisitcs of NL Requirements

During a software development project (mainly when the project leads to the realization of major systems) a fault in requirements specification can determine major delays, costs over-runs, commercial consequences including loss of money, propoerty, layoffs. The achievement of the quality of software requirements is then the first step towards software quality. The process leading to the quality of

requirements starts with the analysis of the requirements expressed in Natural Language (NL) and continues with their formalization and verification (for example by using formal methods).

Despite its inherent ambiguity and informality that determine a difficult proving for correctness, NL is largely used in the software industry for specifying software requirements. Besides the inherent problems of NL, there are other problems which derive from current practices in the industrial SW development process. Due to, for example the volatility of the requirements during the development process and the variable levels of linguistic quality due to the different sources they come from [92], NL requirement specifications are considered as highly risky for software projects [43].

Anyway, the use of NL for specifying requirements indeed has some advantages such as, for example, the ease with which they can be shared among the different people involved in the software development process. In fact, a NL requirement document can be used in different ways, and in different development phases of the final product. For example, it may be used as a working document to be provided as input for architecture designers, testers and user manual editors or it may be also used as an agreement document between customers and suppliers or as an information source for the project manager [89].

In the following a list, yet not exhaustive, of the characterisitcs good-quality requirements are expected to exhibit is provided:

Cohesiveness: each individual requirements should be cohesive, i.e. it shall specify only one thing nad all parts of the requirement belong togheter (i.e. all parts of a data (functional) requirement involve the same data (functional) abstraction, all parts of a quality requirement involve the same quality factor or sub-factor, ...)

Completeness: both an entire requirements specification should be complete and contain all relevant requirements and ancillary material, individual requirements should be complete. This is often a problem because subject matter because subject matter experts who specify requirements often take certain information for granted and omit it, even though it is not obvious to other stakeholders of the requirement.

Consistency: because collections of inconsistent requirements are impossible to implement, individual requirements should be consistent. The consistency can be internal (i.e. among the constituents parts of each requirements, e.g. compound preconditions and postconditions) or external (i.e. between each requirement and its documented sources such as higher-level goals and requirements, and among requirements e.g. two requirements should neither be contraddictory nor describe the same concepts using different words);

Correctness: individual requirements shall be semantically and syntactically correct. A requirement is semantically correct when it meets all or part of an actual need of its relevant stakeholder(s), it is an accurate eleboration of a documented business objective or an higher –level requirement, or all numbers associated with iit have correct values. A requirements is syntactically correct when it expresses imperative sentences (by means of the use of "shall", "must" verbs rather than "will" or "may") and it respects the grammatical rules of the used language.

Currency: all requirements shall be updated when requirements changes in order to avoid they become obsolete. They are also frequently not updated as the architecture is produced, sometimes resulting in changes in the underlying requirements. Both of these problems take testing and maintenance much more difficult.

External Observability: requirements should not unnecessary specify the internal architecture and design of an application or component. Thus, individual requirements should only specify behaviour or characteristics that are extenally observable.

Feasibility: requirements are of no value if the development team cannot implement them. Thus, individual requirements should be feasible given all relevant constraints. It should be possible to implement them given the existing hardware or software technology, the endeavor's budget, schedule and constraints on staffing).

Lack of Ambiguity: individual requirements for an application or component should never be ambiguous. Even if the requirements is intended to be hoghly reusable and therefore general, it should be unambiguous although it may have precise flexibility points. This characteristic is very critical (and often missing) because ambiguous requirements are subject to misinterpretation and are inherently not verifiable. To be unambiguous a requirement should exibit particular properties as, for example: its

meaning should be objective rather than subjective, it shuold be concise, it should have an unique interpretation, it should be understandable to the its intended audiences, it should use specific concrete terms, it should avoid the use of inherently ambiguous terms, whenever possible and practical it should be specified in a quantitative manner.

Mandatory: although requirements can and should be prioritized to help negotiate and schedule them, individual requirements should, by their very nature, be mandatory. Each individual requirement should be essential for the success of the application or component, it should be truly required by some stakeholder, typically the customer or user organization, it should spefify a "what" rather than a "how".

Metadata: individual requirements should have metadata (i.e. attribute and annotations) that characterizes them. This metadata can include (but is not limited to) acceptance criteria, allocation, assumptions, idnetification, prioritization, rationale, schedule, status and tracing information.

Relevance: each requirement should be within the scope of the business, application, or component being specified.

Usability: requirements have many users that use them for different purposes, thus they should be understandable and usable by all of them (e.g. they should be understandable and usable by the managers who must use them for scope control as well as costs, schedule and progess metrics, or by the testers who must verify and validate them).

4.2 Methods for NL Requirements Quality Evaluation

Achieving complete and consistent requirements is in general a chimera. The evaluation of the completeness and consistency has been the object of several works in the past and it is still one of the more challeging field in the software engineering. Several methods, all relying on mathematic notations and formalisms, have been defined. These methods are known as Formal Methods. They can guarantee a formal verification of the consistency and completeness of the requirements. Nevertheless the formal methods are not widely spread in the industry. One of the principal reasons of

this situation is that the definition, establishment and dissemination within a company of formal methods require large investments. Furthermore such rigorous methods should be shared among all the parties involved in a software project, customer included, asking for a further effort.

It is not surprising then that NL is, in spite of its inherent inaccuracies, still the most used technique for representing the requirements.

It is well known that the presence of inaccuracies in requirement documents could introduce serious problems to all the consequent phases of software development. Hence, it is important to provide methods and tools for the analysis of the NL requirement documents [73], [76], [78].

Unfortunately, the state of the art and practice witnesses a lack of tools and techniques for the NL requirements analysis. The following list, yet is not exaustive, includes the most popular practices and tools used in the practice as countermeasures for mitigating the negative effects of the use of NL in requirements specification:

Tools for analysis: in the litterature few descriptions of tools for NL requirements exist. One of the most known tool is ARM. This tool, developed by NASA, although quite simplicistic, is able to perform a lexical analysis for detecting some defects. The defects are mainly identified by means of special terms and wordings that reveal particular defects.

Means for espessing NL requirements: in the practice several means and techniques have been defined in order to mitigate the inherent ambiguity of NL. The most common is the use of templates of structuring requirements documents or the adoption of a restricted english (avoiding ambiguous terms and styles) for expressing the requirements. A way to define requirements that in recent years is going to be spread in the industry are the Use Case [13], [14]. This way to express requirement allows a functional description of the requirements and imposes a (light) formalism based on the NL to the requirements.

Practices for mitigating the effects of the NL inherent ambiguity: because no technique nor tool can guarantee the absence of ambiguities in a NL requirements, in the practice some countermeasures are frequently adopted. For example, joint reviews of the requirements documents are preformed by customers and suppliers together.

The aim of these joint reviews is to verify that the different developers and the customers have the same understanding of each requirement. For doing that the use of glossaries may be of great help.

Several studies dealing with the evaluation and the achievement of quality in natural language requirement definition can be found in the literature. We will briefly discuss some of those we consider to be of particular interest.

Macias and Pulman [78] apply domain-independent NLP techniques to control the production of natural language requirements. They propose the application of NLP techniques to requirements documents in order to control:

the vocabulary used, which must be fixed and agreed upon,

the style of writing, i.e., a set of pre-determined rules that should be satisfied in order to make documents clear and simple to understand; they associate an ambiguity rate to sentences, depending on the degree of syntactic and semantic uncertainty of the sentence, the information conveyed by requirements, by discovering underspecifications, missing information, unconnected statements.

Finally, they discuss how NLP techniques can help the design of subsets of the English-grammar to limit the generation of ambiguous statements

Goldin and Berry [46] implemented a tool for the extraction of abstractions from natural language texts, i.e. of repeated segments identifying significant concepts on the application field of the problem at hand. The technique proposed is restricted to a strict lexical analysis of the text.

Hooks [51] discusses a set of quality characteristics necessary to produce well-defined natural language requirements. This paper presents some common problems which arise when requirements are produced and looks at how to avoid them. It provides an in depth survey of the principal sources of defects in natural language requirements and the related risks.

Wilson and others [107], [108] examine the quality evaluation of natural language software requirements. Their approach defines a quality model composed of quality

attributes and quality indicators, and develops an automatic tool to perform the analysis against the quality model aiming to detect defects and collect metrics.

Other works investigate how to handle ambiguity in requirements. In particular, Fuchs [44] proposes to solve the problems related to the use of NL in requirements documents by defining a limited natural language, called Attempt Controlled English (ACE), able to be easily understood by stakeholders and by any person involved into the software development process and simple enough to avoid ambiguities allowing domain specialists to express requirements using natural language expressions and to combine these with the rigour of formal specification languages.

Kamsties and Paech [69] focus especially on the ambiguity evaluation of natural language requirements. They start from the consideration that ambiguity in requirements is not just a linguistic-specific problem and put forward the idea of a checklist addressing not only linguistic ambiguity but also the ambiguity related to a particular domain.

Mich and Garigliano [81] put forward a set of measures for semantic and syntactic ambiguity in requirements. Their approach is based on the use of information on the possible meanings and roles of the words within a sentence and on the possible interpretation of a sentence. This is done using the functionalities of a tool called LOLITA.

Natt och Dag et alt. [83] recently presented an approach based on statistical techniques for the similarity analysis of NL requirements aimed at identifying duplicate requirement pairs. This technique may be successfully used for revealing inter-dependencies and then may be used as a support for the consistency analysis of NL requirements. In fact, the automatic determination of clusters of requirements dealing with the same arguments may support the human analysis, aimed at detecting inconsistencies and discrepancies, by focusing on smaller sets of requirements.

4.3 A NL Requirements Quality Model

The first step of a quality evaluation of any entity is to define a Quality Model against which it will be evaluated. In our case, a Quality Model against wich NL requirements

could be evaluated from a linguistic point of view in order to identify and possibly remove ambiguities, inconsistencies and incompletenesses has been defined. The Quality Model is composed of quality properties to be evaluated by means of quality indicators.

Due to the inherent ambiguity originating from different interpretations of NL descriptions, the use of NL as a way to specify the behavior of a system is always a critical factor.

The Quality Model for NL requirements defined in this thesis has been conceived to be used as a basis for the development of a tool that, relying on linguistic techniques for the analysis of NL texts, can be envisaged also to remove interpretation problems in NL requirements documents. The analysis made by means of NL-based techniques is useful to address several interpretation problems related to linguistic aspects NL requirements. These problems may be grouped into three main categories:

***Expressiveness*: it includes those characteristics dealing with a incorrect understanding of the meaning of the requirements. In particular, the presence of ambiguities in and the inadequate readability of the requirements documents are frequently causes of expressiveness problems.**

***Consistency*: it includes those characteristics dealing with the presence of semantics contradictions in the NL requirements document.**

***Completeness*: it includes those characteristics dealing with the lack of necessary information within the requirements document.**

The application of linguistic techniques to NL requirements, allows their analysis from a lexical, syntactical or semantic point of view. For this reason it is proper to talk about, for example, lexical non-ambiguity or semantic non-ambiguity rather than non-ambiguity in general. For instance, a NL sentence may be syntactically non-ambiguous (in the sense that only one derivation tree exists according to the syntactic rules applicable) but it may be lexically ambiguous because it contains wordings that have not a unique meaning.

Figure 4.1 shows schematically that the quality of NL requirements can be represented as a two-dimensional space, where the horizontal dimension is composed

of the main target qualities to be achieved (Expressiveness, Consistency and Completeness) and the vertical dimension is composed of the different points of view from which the target qualities can be considered.

The difficulty level of application of linguistic techniques varies according to the kind of analysis: in fact, while it is relatively easy to perform lexical analysis, it is much harder to face semantic problems in NL requirement documents as they are. At lexical level only the single wordings used in the sentences are considered, while at the syntactical level it taken into account also the syntactical structure of the sentences (i.e. it has to take into account also the role that each term plays in the sentneces). At the semantic level there is the need to derive the semantics of the sentences (i.e. the meaning of the whole sentences).

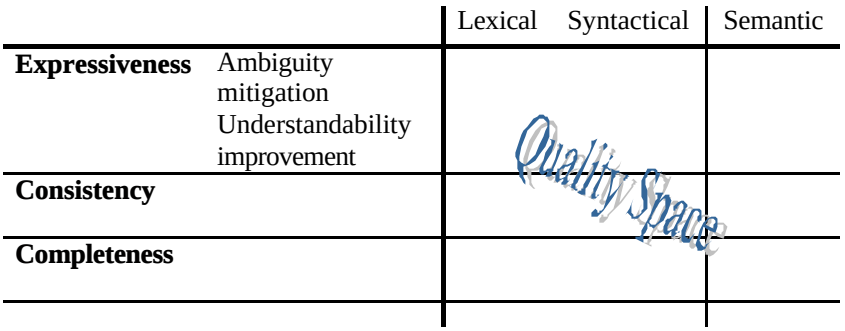


Figure 4.1. Two-dimensional representation of the NL requirements quality

Linguistic techniques can effectively address the issues related to the Espressiveness because the lexical and syntactical levels provide means enough to obtain effective results. For this reason the Quality Model described in this section addresses the Expressiveness property of NL requirements and it doesn't take into consideration the Consistency and Completeness properties. These properties are anyway considered in this work as shown in the section 4.4.

Because, as any other evaluation process, the quality evaluation of NL software requirements has to be conducted against a Model. The Quality Model we defined for the natural language software requirements is aimed at providing a way to perform a quantitative (i.e. that allows the collection of metrics), corrective (i.e. that could be

helpful in the detection and correction of the defects) and repeatable (i.e. that provides the same output against the same input in every domains) evaluation.

The quality model we defined is composed of three high-level quality properties for NL requirements to be evaluated by means of indicators directly detectable and measurable on the requirement document.

The higher level properties of the Quality Model are:

***Unambiguity*: the capability of each Requirement to have a unique interpretation.**

***Specification Completion*: the capability of each Requirement to uniquely identify its object or subject.**

***Understandability*: the capability of each Requirement to be fully understood when used for developing software and the capability of the Requirement Specification Document to be fully understood when read by the user.**

Indicators are syntactic or structural aspects of the requirement specification documents that provide information on defects related to a particular property of the requirements themselves. Tables 4.1, 4.2, 4.3 describe the Indicators related to each Quality Property along with examples of the keywords to be used for detecting potential defects in the NL requirements.

Unambiguity Property	
Indicator	Description
Vagueness	It is pointed out when parts of the sentence hold inherent vagueness, i.e. words having a non uniquely quantifiable meaning
Subjectivity	It is pointed out if the sentence contains wordings used to express personal opinions or feeling
Optionality	It is pointed out if the sentence contains an optional part (i.e. a part that can or cannot considered)
Implicity	It is pointed out in a sentence when the subject or the object is generically expressed

	subject or the object is generically expressed
Weakness	It is pointed out when a sentence contains a "weak" verb

Table 4.1 Ambiguity Indicators

Specification Completion Property	
Indicator	Description
Under-specification	It is pointed out when the sentence contains a word identifying a class of objects without a modifier specifying an instance of this class

Table 4.2 Specification Completion Indicators

Understandability Property	
Indicator	Description
Multiplicity	It is pointed out if the sentence has more than one main verb or more than one subject
Readability	The Coleman-Liau Formula readability metrics: (5.89*chars/wds-0.3*sentences/(100*wds)-15.8]). The reference value of this formula for an easy-to-read technical document is 10, if it is >15 the document is difficult-to-read

Table 4.3 Understandability Indicators

The proposed Quality Model has been defined with the aim to detect and point out potential syntactic and semantic deficiencies that can cause problems when a NL requirements document is used or is transformed in a more formal document. The definition of the criteria used in the Quality Model has been driven by some results in the natural language understanding discipline, by an experience in formalization of software requirements and also by a depth analysis of real requirements documents industrial partners provided with. Moreover this quality model has been defined after a study of the existing related literature and by taking advantage from matured experience in the field of requirement engineering and software process assessment according to the SPICE (ISO/IEC 15504) model (see chapter 3). This quality model,

though not exhaustive, is sufficiently specific to include a significant part of lexical and syntax-related issues of requirements documents.

The defined quality model does not cover all the possible quality aspects of software requirements but it is sufficiently specific for being applied (with the support of an automatic tool) for comparing and verifying the quality of requirement documents.

The sentences recognized as defective according to the quality model described in Tables 4.1, 4.2, 4.3 are not uncorrected sentences in terms of English Language rules. Rather they are incorrect in terms of the above defined expressiveness characteristics.

The quality model has been derived taking into account its principal purpose: its should be intended as a starting point for the realization of an automatic tool for the analysis of NL requirements. The indicators the quality model is composed of are terms and linguistic constructions characterising a particular defect and being directly detectable looking at the sentences of a requirements document.

For this reason in Table 4.4 some notes explain how the Indicators beloning to the quality model can be pointed out by perfoming a linguistic analysis of the requirements document:

Indicator	Notes
Vagueness	The occurence of this Indicator is due to the existence of Vagueness-revealing wordings as for example: clear, easy, strong, good, bad, useful, significant, adequate, recent, ...
Subjectivity	The occurence of this Indicator is due to the existence of Subjectivity-revealing wordings as for example: similar, similarly, having in mind, take into account, as [adjective] as possible, …
Optionality	The occurence of this Indicator is due to the existence of Optionality-revealing words as for example: possibly, eventually, if case, if possible, if appropriate, if needed, …
Implicity	The occurrence of this Indicator is determined by the existence of: Subject or complements expressed by means of: Demonstrative adjective (this, these, that, those)

	or Pronouns (it, they…)or Terms having the determiner expressed by a demonstrative adjective (this, these, that, those) or implicit adjective (as for example previous, next, following, last...) or preposition (as for example above, below...)
Weakness	The verbs that determine the occurrence of this indicator are the Weak verbs: could, maight, may.
Under-specification	This Indicator occurs when words needing to be instantiated are found, for example: flow (data flow, control flow, ..), access (write access, remote access, authorized access, ..), testing (functional testing, structural testing, unit testing, ..), etc.
Multiplicity	This indicator occurs when a multiple sentences is found
Readability	It correspond to the actual value of the Coleman-Liau formula

Tablke 4.4 Explanatory Notes

4.4 The Liguistic Approach to the NL Requirements Consistency and Completeness Evaluation

What is the extent linguistic techniques able to provide an effective support in the consistency and completeness analysis of NL requirements documents? For answering this question we investigated the possibility to provide techniques and tools able to effectively support this kind of analysis without adopting Formal Methods.

Consistency and completeness analysis of NL requirements shall begin with the identification of the items addressing related objects in order to understand if they contain inconsistencies or/and incompletenesses. Then, it is necessary, as first step of consistency and completeness analysis, to put togheter all the senteneces dealing with specific topic. We call such as group of sentences View. Possible topics are:

quality characteristics (attributes) of the product described in the document under analysis (e.g. security, efficiency, …);

components of the products described in the requirements document or belonging to the external environment (e.g. the user interface or the user);

functionalities of the product (e.g. the printing function);

The derivation of a View from a document relies on the availability of special sets of terms representing key words related to the topic the View is related to. We call these sets of terms V-dictionaries.

The derivation of the Views can be automatically made by using NL understanding techniques. In the following the defined methodology to derive Views from a NL requirements document is described. This methodology is based on the existence the V-dictionaries of interest containing wordings that can be put in relation with a particular topic.

The construction of these V-dictionaries is done by the user and it relies on his skill and on the study of appropriate (technical) documentation.

Once the V-dictionaries of interest have been built, the identification of those sentences belonging to the related View can be made automatically by relying on the output of a syntax analysis and on the appropriate V-Dictionary. In fact, those sentences having the subject or the object expressed by terms belonging to the V-Dictionary can be tagged as beloniging to that View. In other words the idea is to put togheter those sentences in the requirements document directly or indirectly dealing with a particular topic.

The output of this analysis can be used to support the consistency and completeness analysis of a requirements document, because the person that performs the analysis can concentrate his work on sub-sets of the sentences of the document. That can make easier the detection of possible problems.

One of the principal advantages in applying the View approach for supporting the consistency and completeness analysis of a NL requirements document is the capability to detect misplaced sentences. A misplaced sentece, in this case, is a sentece dealing with a particular aspect of the system described by the requirements but not included in the part of the document where that aspect is trated. An example is provided in Figure 5.9. The figure shows a View derivation made an a exempler

document taken form an industrial project. The View in this case is the “security” characterisitc, then the graph represents the position of all the sentences dealing with security over the requirements document. As the figure idicates, the View is composed of nine senteces, then the security requirements are supposed to be within these senteces. Seven of them belong to Section 3 the other two to Section 2. Section 3 is titles “Safety and Security requirements”. The other two security requirements found in Sectioin 2 represent the added value given by the tool. In fact, if the analyser of a requirements document would perform a consistency or completeness analysis of the security requirements it can be expected that his attention was concentrated on section 3. The derived View says to the analyser that it is necessary to take into account also the two of section 2 and this could not to be so evident for the user without the tool.

The technique described above for identifying those sentences belonging to a view is indeed a difficult and challenging one. In particular, the effectiveness and the completeness of the Views strictly depends on the associated V-dictionary: the more the V-Dictionary is precise and complete, the more the effectivenress of the outcomes increases. Depending on the technique used and on the content of the V-dictionary, it is possible that the V-Dictionary misses some relevant terms. It could be possible that some terms generally relating to a View have been omitted or that the document under analysis contains specific terms that only in its context can be considered relating to that View.

As an example of use of this technique, if the view of interest is related to the security quality characteristic of the product, it may be possible that in the document such a sentence occurs: *The firewall to be used in the system is Apache*. It can be expected that from there in after the firewall (that is certainly a term relating to security) will be indicated by means of the term Apache. Nevertheless, because Apache is the name of a specific tool, we can expect that it could be not included in the V-Dictionary relating to security. For solving this kind of problems a new methodology for enriching the set of terms based on syntactical analysis of the document under analysis can be implemented.

This methodology relies on the concept of Subject-Action-Object (SAO) triplet. As such triple can be easily derived from the output of the syntactic parser.

The idea is to select the SAO triplets having the Action identified by a special verb. We might start with a ad-hoc V-Dictionary, then consider, for instance, the following verb categories: Compositional verbs (e.g. to comprise, to compose, to include, …), Functional verbs (e.g. to support, to include, to use, …) and Positional verbs (e.g. to follow, to precede, ..), and proceed this way: If either the Subject (the Object) of the SAO belongs to the special set of terms, then also the Object (the Subject) of that SAO can be considered as a candidate to be included into the set of terms. This methodology allows, for instance, to include in the security related V-Dictionary the term Apache w.r.t. the example above.

Following this approach, in order to further enrich the V-Dictionaries, other related literature (not only the requirements documents) could be analyzed (as for example related technical documents) in order to identify a larger number of terms for the V-dictionary.

5. The Tool QuARS

In this chapter the way the quality evaluation of natural language requirements documents, against the quality model defined in chapter 4, has been made automatic is described. For doing that I designed and developed a tool (called QuARS – Quality Analyzer for Requirements Specification) able to analyze a text document and point out expressiveness defects. QuARS is able also to support the consistency and completeness analysis by means of the derivation of the Views. The architecture and the functional description of the QuARS tool are provided in this chapter along with a discussion of the strenghts an the improvement opportunities of it.

5.1 Introduction

Natural language (NL) is still the most common way to express requirements. The use of NL for specifying requirements indeed has some advantages such as, for example, the ease with which they can be shared among the different people involved in the software development process. In fact, a NL requirement document can be used in different ways, and in different development phases of the final product. For example, it may be used as a working document to be provided as input for architecture designers, testers and user manual editors or it may be also used as an agreement

document between customers and suppliers or as an information source for the project managerement. The principal disadvantage is the risk, because the inherent ambiguity and informality of the NL, to have different interpretation of a requirement.

The availability of methods and techniques for performing an analysis of NL requirements could reduce this risk saving the advantages.

The way that usually in the research community the problem of the requirements is approached is basically moving towards a more formalized way to express them. Formal methods are mathematical-based representations of the requirements. The advantages that can be obtained by means of these methods have some costs in terms of the necessity of skilled people for the definition and also for the use of them. Furthermore, the communication mechanism between all the stakeolders (customer included) may be difficult because not all of them can be able to understand and manage such a requirements formalism.

When a formal method is applied the initial requirements document is always written in NL and from it the formal specification is derived (see figure 5.1). It is then evident that possible defects in terms of ambiguity in the initial NL document can be moved into the formal version of requirements. The passage from the initial informal representation of requirements and the (first) formal representation of them is the most critiacl point when formal methods are adopted. Performing an analysis of NL requirements to detect and remove ambiguity defects is of interest also to reduce the gap between the NL representation of requirements and the following representation made with formal methods.

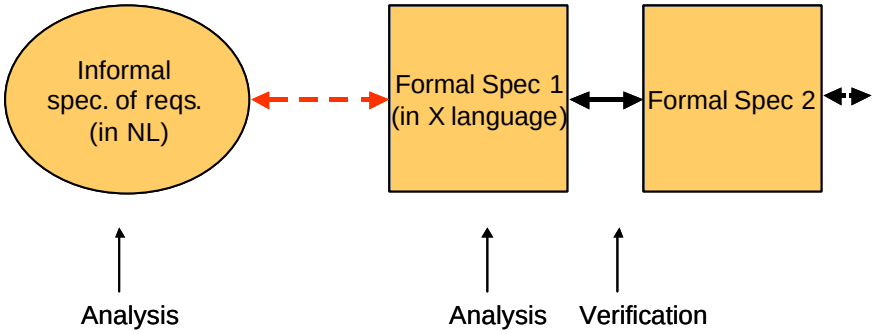


Figure 5.1: The requirements formalization process

5.2 Linguistic Techniques for Defects Detection

Starting from the quality model defined in the chapter 4, a feasibility analysis for the different kinds of defects is made, and what is necessary to implement the automatic evaluation is described

The principal objective of my research activity has been to provide a concrete support for the software practictioners. Because in the practice NL is the most used mean for expressing requirements and because there is a lack of supporting tools and techniques for the analysis of this kind of requirements, my research activity has been conducted with the aim to develop an automatic tool able to analyze NL requirements "as they are", i.e. without move towards an other formalism.

After the definition of the Quality Model for NL requirements the further phase of the research activity I performed was the design and implementation of and automatic tool able to detect the lacks of quality in a NL requirements document according to the Quality Model. This tool should also provide a support for the completeness and consistency analysis.

The Quality Model contains a set of Indicators, that are linguistic elements of the sentences of a NL requirements document that express a potential defect. In order to make automatic the detection of these indicators within a document, different linguistic techniques can be used.

The techniques that can be used may belong to two different cathegories:

Lexical techniques

Syntactical techniques

The lexical techniques rely on the simple reading and recognition of the terms of the sentences belonging to the requirements document. The kind of analysis that can be performed by means of the lexical techniques is a morphological analysis, i.e. it is possible to verify that the single terms occurring in the sentences are correctly written (according to the english lexicon) or suitably choosen.

The syntactical techniques are more sophisticated as respect the lexical ones. The syntactical techniques are based on the knowledge of the syntactical relations accurring among the different terms of a sentence. A syntactical technique can allow the analysis of a requirements document relying on the knowledge of the syntactic roles of each item of a sentence and of the relations among them (i.e. it is possible to know what is the subject, the related verb and the associated complements).

Some of the Indicators of the Quality Model can be pointed out by applying lexical techniques others need the application of syntactical techniques for beeing detected.

In the Table 5.1 the correspondence between each Quality Model Indicator and the necessary technique to be applied to automatize its detection is shown.

Indicator	Lexical technique	Syntactical Technique
Vagueness	X	
Subjectivity	X	
Optionality	X	
Implicity		X
Weakness		X
Underspecification		X

Multiplicity		X
Readability	X	

Table 5.1: Quality Model Indicator vs. Linguistic Techniques

The analysis of NL requirements documents should not be limited to the detection of the Indicators of the reference Quality Model. It should be automatized the derivation of the information for supporting the consistency and completeness analysis too. As said in chapter 4, the View derivation is a way to provide the requirements analyser with a practical supoort for this kind of analysis. The derivation of the Views can be basically made by means of Lexical techniques, but, for not beeing limited to the mere detection of a the occurrences of the terms belonging to a domain dictionary the application of syntactical techniques is necessary.

5.3 Design of an Automatic Tool for NL requirements evaluation

In this section the high architectural description of the tool is provided. The development of the tool has been driven by the objective to be mainly modular, extesible and usable. The acritectural design matches the fist two characterisitcs. In Figure 5.2 the high level architectural design is depicted.

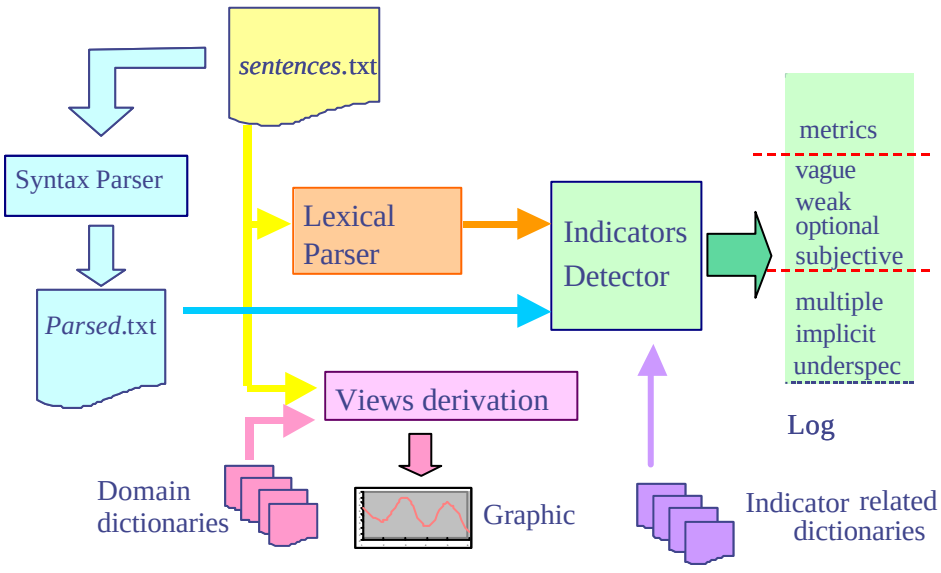


Figure 5.2: High level architectural design

The main components of the tool are the syntax parser, the lexical parser, the view derivator.

5.3.1 Syntax Parser

This component derives the syntactical structure of each sentence contained in the requirements document. The used application used to do that is Minipar [82]. This application associates tags to the terms of the sentence, these tags indicate the syntactical role of each of them. The relations among the syntactical components of the sentence are derived too. As an example the sentence:

`1. The system shall provide the manual of the user`

Is syntactically analysed by the syntax parser in the following way:

```
> (
1    (the ~ Det2    det)
2    (system    ~ N  4    s)
3    (shall     ~ Aux4    aux)
4    (provide   ~ V  E0   i    (gov fin))
5    (the ~ Det6    det)
6    (manual    ~ N  4    obj)
7    (of  ~ Prep    6    mod)
8    (the ~ Det9    det)
9    (user~ N  7    pcomp-n)
)
```

This outcome has to be interpreted as follows:
line 1: the term `the` is the determiner (tag `Det`) of the noun `system` at line number 2 (tag 2)

line 2: `system` is a noun (tag `N`) and it is the subject (tag `s`) of the verb at line 4 (tag `4`)
line 3: `shall` is the auxiliary (tag `Aux`) of the verb at line 4 (tag `4`)
line 4: `provide` is the main verb (tag `V`)
line 5: `the` is the determiner of the noun at line 6 (tag `6`)
line 6: `manual` is a noun (tag `N`) playing the role of object (tag `obj`) of the verb at line 4 (tag `4`)
line 7: `of` is a preposition (tag `Prep`) of the term at line 6 (tag `6`) and it plays the role of modifier of it (tag `mod`).
line 8: `the` is the determiner (tag `Det`) of the term at line 9 (tag `9`).
line 9: `user` is a noun (tag `N`) playing the role of complement (tag `pcomp-n`) due to the term of at line 7 (tag `7`)

The Minipar syntax parser is one of the most spread English language syntactical parsers. A syntax parser, on the basis of the rules of the English language calculates one of the possible derivarion tree for the sentence under analysis. Because it could be possible that, applying the English language rules, more than one derivation tree exists for the same sentence, the Minipar parser may provide a wrong syntax recognition of a sentence. This problem is common to all the exisitng syntax parser for English sentences, but Minipar guarantees an higher rate of correctly derived sentences: about the 85%.

5.3.2 Lexical Parser

This components is based on the identification of the single terms appearing in the sentences belonging to the requirements document. This component is used to perform a morphological analysis of the sentences and for supporting those kind of analysis based on the detection of the occurrences of special terms or wordings in the requirements document. The lexical analyzer relies on the WordNet English Dictionary.

5.3.3 Indicators Detector

This original component points out, on the basis of the outcomes of the syntax and lexical parsers, the occurrences of the indicators in the single senteces of the document

under analysis and writes them in the correspondent log file. This component along with the View derivator, described in Section 5.3.4 have been fully developed, using the C++ language, during my PhD course.

5.3.4 View Derivator

This original component, acquires, as first step, the structure of the document in terms of sections and sub-sections partition. Then performs the recognition of a sentence as belonging to a View. It consider a sentence as belonging to a View according to the syntactical rules defined on section 4.4. Finally, it counts the number of sentences recognized as belonigng to a View occurring in each (sub-) section of the requirements document. These data are graphically represented as output.

5.3.5 Dictionaries

Dictionaries are the passive components of the tool. They contain sets of terms that are necessary to perform syntactical, lexical analysies and View derivations. The number and the content of these dictionaries may vary according to the application domain and user needs.

5.3.6 Input and Output

The input of the tool is composed of:

the requirements document to be analyzed. The allowed format of this input file is simple text (.txt, .dat, ... format). This file is given to the syntax parser component that produces a new file containing the parsed version of its sentences according to the format described in section 5.2.1;

the indicator-related dictionaries. They may contain either the terms indicating a kind of defects according to the quality model or the domain dictionaries to be used for the View derivation. The dictionaries have to be in simple text format.

The Output of the tool are:

the log files containing the indications of the sentences containing defects. A log file for each kind of analysis is produced.

the calculation of metrics about the defect rates of the analyzed document.

the graphical representation of the Views over the whole analyzed document.

5.4 Functional description of QuARS

In this section the functional description of the tool developed is provided. The way this description is provided is by means of pictures of the QuARS’s Graphical User Interface (GUI). The user interface has been developed using the TCL-TK language. In figure 5.3 the GUI when the tool is in its in the initial state is shown.

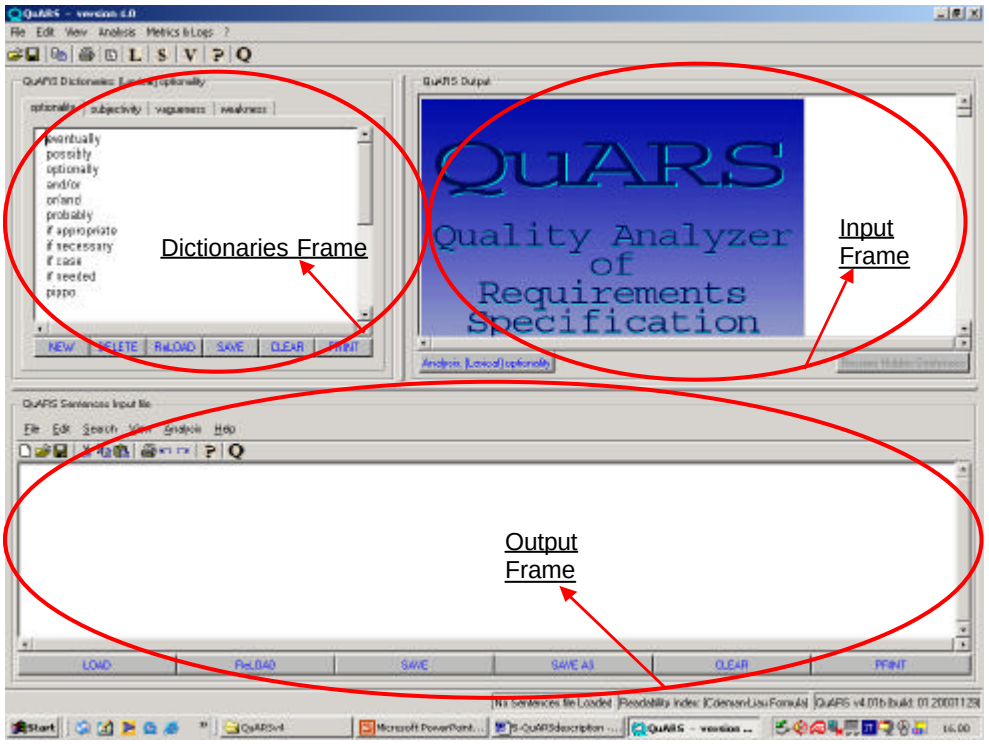


Figure 5.3: Principal frames of the QuARS GUI

It is composed of three principal frames:

the Dictionaries frame: the content of the dictionaries is shown here, along with some function buttons for the dictionary handling.

the Input frame: the content of the text file containing the requirements to be analyzed is shown here. Some function icons and buttons are provided in this frame, they allow the loading, the hadling and the saving of the input file.

the Output frame: this frame contains the outcomes of the analysis.The principal fuctionalities of the QuARS tool are: requirements expressiveness analysis, support for requirements consistency and completeness analysis, metrics derivation. In the following they are described in detail.

5.4.1 Expressiveness Analysis

In this section how QuARS performs the expressiveness analysis is described. The expressiveness analysis aims at detecting defects in the requirements that could lead to misinterpretation problems. The expressiveness analysis is conducted against the quality model described in chapter 4. Both lexical-based analysis and syntax-based analysis are used to implement this kind of analysis.

5.4.1.1 Lexical-based analysis

With reference to figure 5.4, for performing one of the lexical-based analysies it is necessary to select the "L" button on the top tool bar of QuARS (arrow 1).

Once the lexical-based analysis has been selected, in the Dictionaries frame the dictionaries corresponding to all the available lexical-based analysies are shown (arrow 2). It is possible to select the kind of analysis of interest by selecting the correspondent dictionary book-mark in the Dictionaries frame. The primitive available Dictionaries for the lexical-based expressiveness analysis are: optionality, subjectivity, vagueness and weakness.

Before starting the analysis the text file containing the requirements has to be loaded. For doing that the LOAD button in the Input frame has to be selected. A window for

the selection of a text file over the whole PC file system appears. Once the input file has been selected, its content appears in the Input frame.

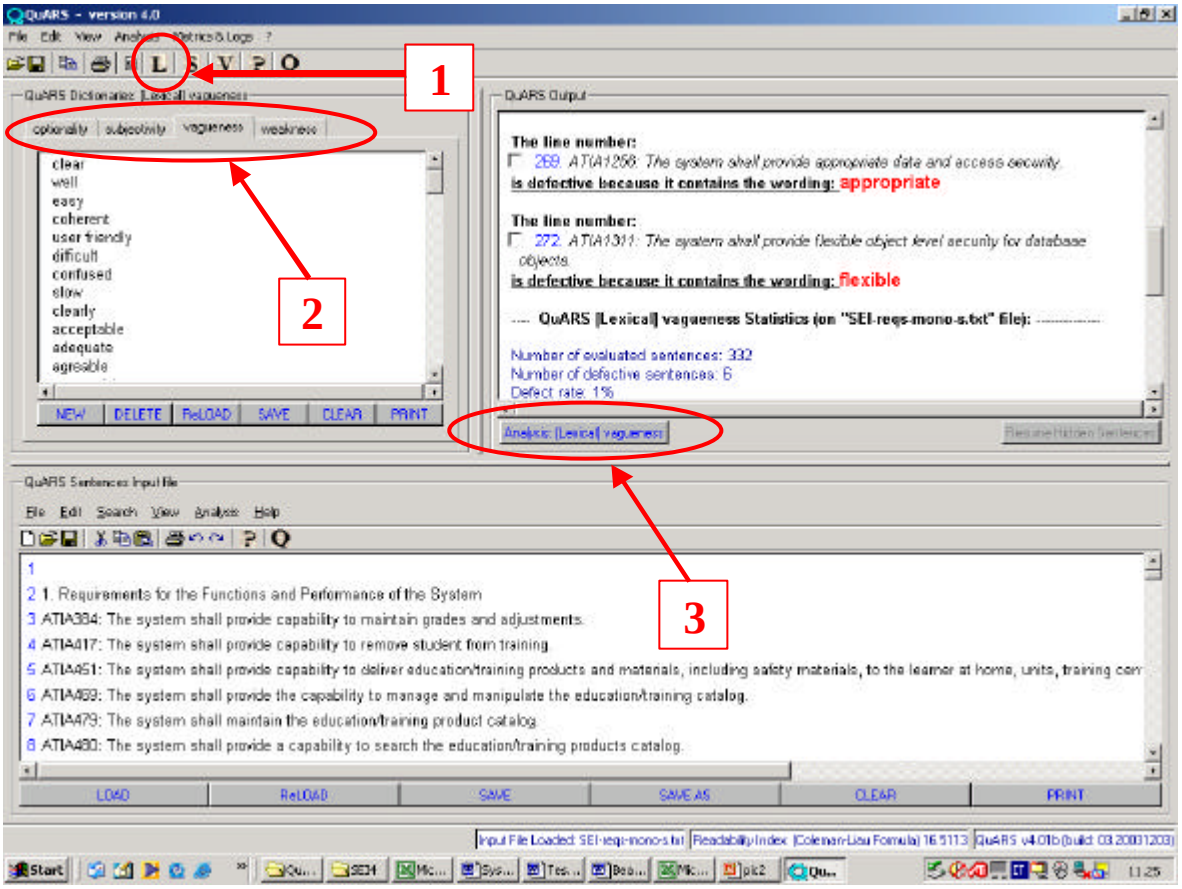


Figure 5.4: Lexical-based Expressiveness anlysis

The analysis starts when the Analysis button is pushed (Figure 5.4 – arrow 3). In the Output frame those sentences belonging to the file under analysis containing the particular defect we are investigating on are shown along with the indication of the individual term that makes the sentence defective (according to the kind of analysis selected).

The defective sentences can be now corrected. It is possible to point out each defective sentence directly on input file. If a sentence in the Output frame is clicked, the same sentence in the input file is highlighted in the Input Frame (see Figure 5.5). The user is now able to modify the defective sentence for correction in the Input frame. After the defective senteces have been corrected the analysis can be re-done in order to verifiy that no new errors have been introduced.

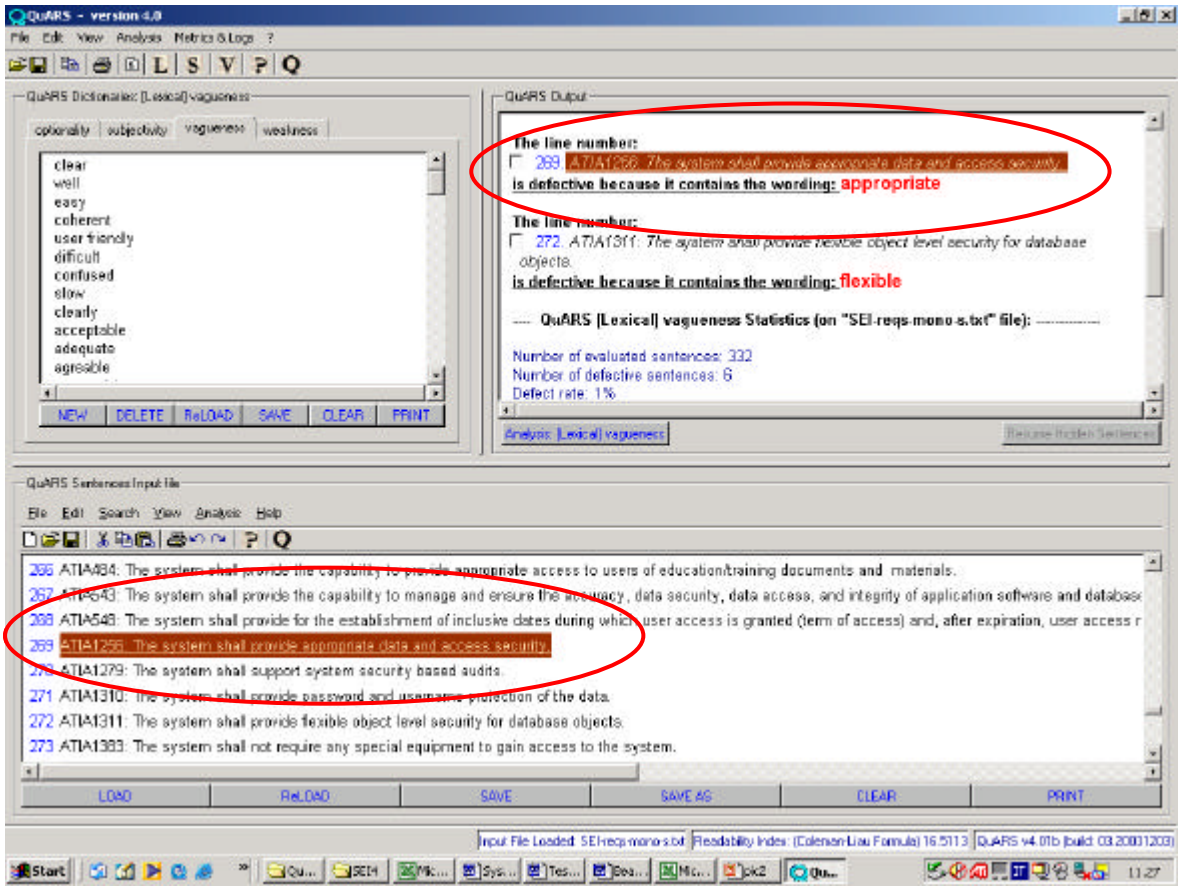


Figure 5.5: Active link between the Output and the Input frames

It could be possible that the tool points out a "false positive". A "false positive" is a sentence recognized as defective by the tool but considered acceptable by the user and then it doesn't need any corrective action. In this case the user can activate the correspondent check button (Figure 5.5 - arrow 1). The "false positive" sentences will be hidden in the output frame in order to do not display them any more (even if the tool still considers them defective) (see Figure 5.6). The hidden sentences can be displayed again by clicking the "Resume Hidden Sentences" button in the Output frame.

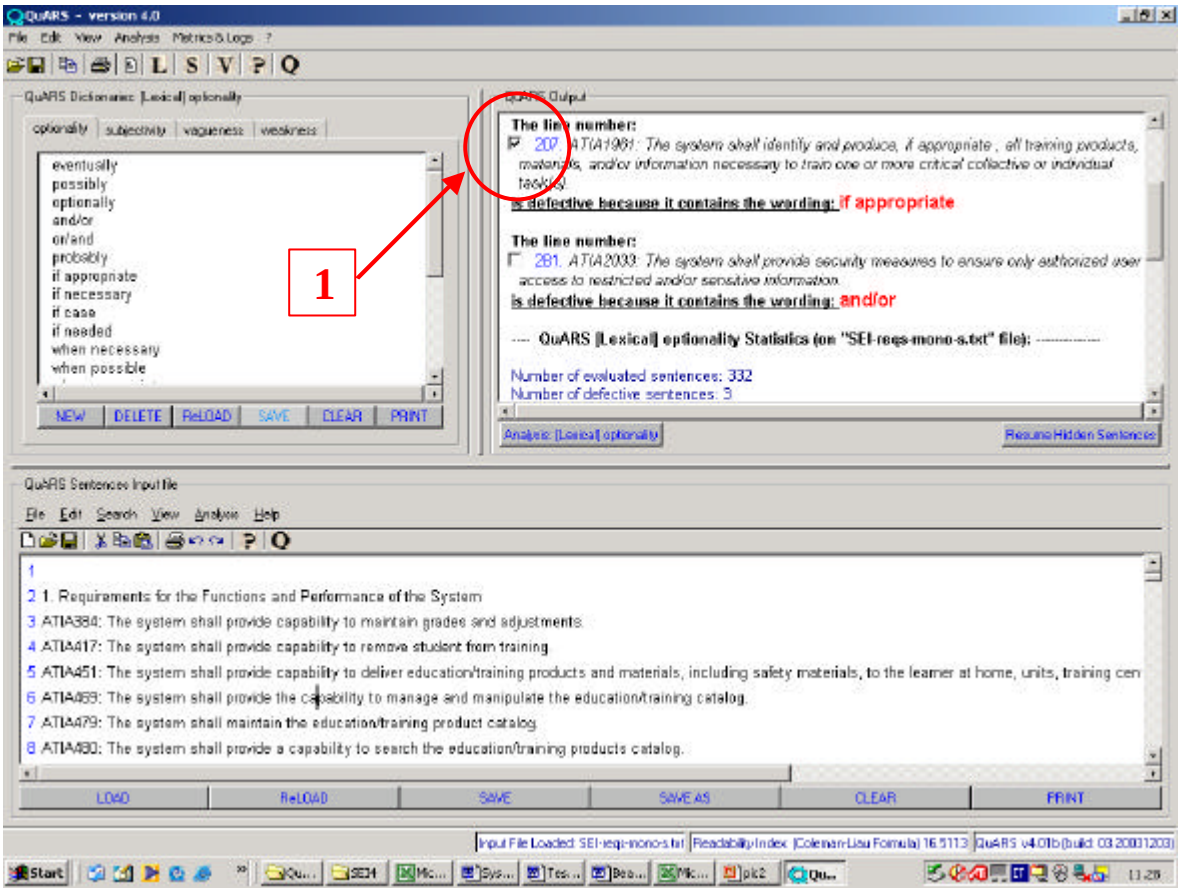


Figure 5.6: Hiding the "false positive"

5.4.1.2 Syntax-based Analysis

With reference to Figure 5.7, for performing the syntax-based expressiveness analysis the "S" button on the top tool-bar of the QuARS GUI (arrow 1).

Once the syntax-based analysis has been selected, in the Dictionaries frame the possible dictionaries of each type of the available syntax-based analysies are shown (arrow 2). It is possible to select the type of analysis of interest by selecting the correspondent dictionary book-mark in the Dictionaries frame. The available analysies are: Implicity, Multiplicity and Underspecification.

The way the analysis is performed is the same than the lexical-based analysis. The difference between the two types of analysies is transparent to the user. In fact, for performing these analysies the tool, first derive the syntactical structure of easch sentence in the input file. Anyway, the syntactical structure of the sentences is not displayed. This structure is necessary for detecting the implicity, multiplicity and

underspecification defects in the requirements document. While for the implicity and multiplicity analysis dictionaries are necessary, the multiplicity analysis, even though it relies on the syntax structure of the sentences, for its nature, it doesn't need any dictionary.

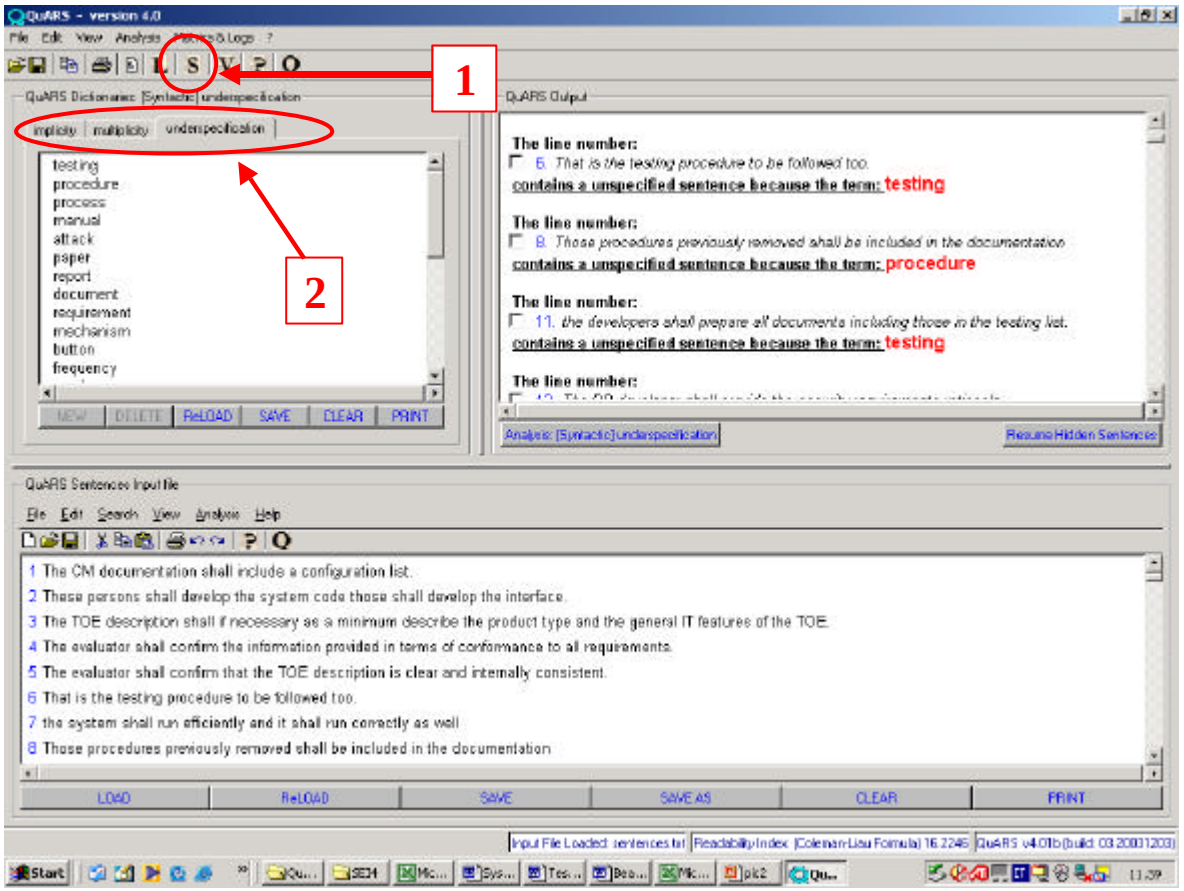


Figure 5.7: Syntax-based analyses

5.4.2 Consistency and Completeness Support

The functionality of the QuARS tool for supporting the consistency and completeness analysis is the View derivation. In this case, diffently from the expressiveness analysis, the aim is not the detection of defects in the sentences of the requirements document under analysis, but the derivation of semantic information that can be of help in this kind of analysis.

The semantic information is basically the “argument” a sentence deals with. A View is essentially a filter for those senteces dealing with a particular argument. The tool QuARS is able to graphically show the number of occurences of sentences belonging to a particular View for each section the requirements document is composed of. The View derivation relies again on dictionaries. The V-Dictionaries contain domain-related terms (insead of defect-related terms). A V-Dictionary is the sets of “all” the terms dealing with a particular View, i.e. those terms that, if contained in a sentence, indicate that this sentence is dealing with the argument the View is referring to.

In this section the way QuARS derives this information is described again by means of pictures of its GUI.

With reference to Figure 5.8, for selecting the View derivation fuctionality of the QuARS tool the “V” button on the top tool-bar of the GUI has to be clicked (arrow 1).

Once the View derivation functionality has been selected, in the Dictionaries frame the available dictionaries are shown (arrow 2). Each dictionary in the Dictionaries frame corrsponds to a particular View that can be derived. It is possible to select the View of interest by selecting the correspondent V-Dictionary.

Once the requirements file is loaded, the View derivation can be started.

The output of the View derivation is a table, displayed in the Output frame,the rows of this table correspond to the (sub-)sections of the document under analysis. The first column contains the number of sentences belonging to the View and the second column the total number of sentences of the correpsondent (sub-)section.

These data are graphically shown by means of a MS Excel graph corresponding to the table.

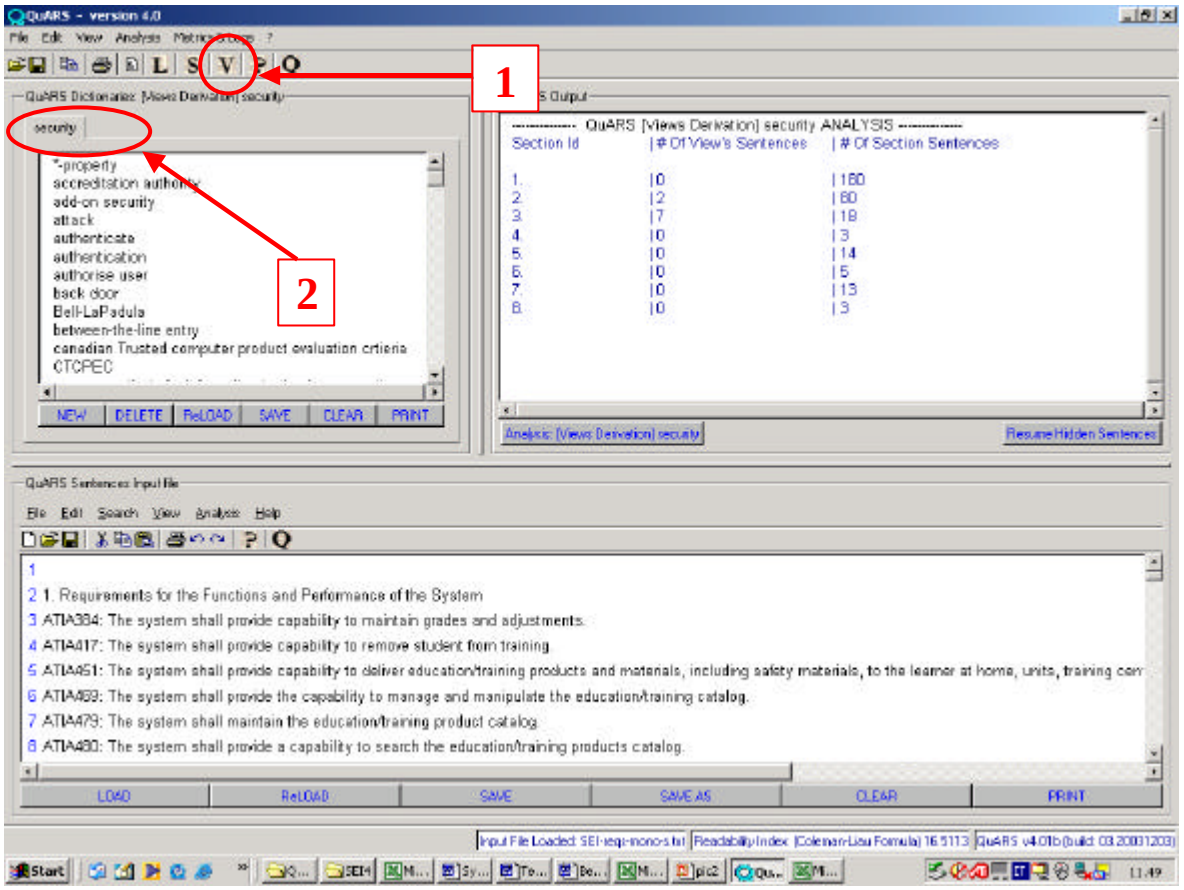


Figure 5.8: View derivation

In Figure 5.9 the output of the View derivation id depicted. In this case, the View derived is the security characteristic. In fact, the dictionary in the Dictionaries frame contains a set of terms directly or indirectly correlated to a security issue.

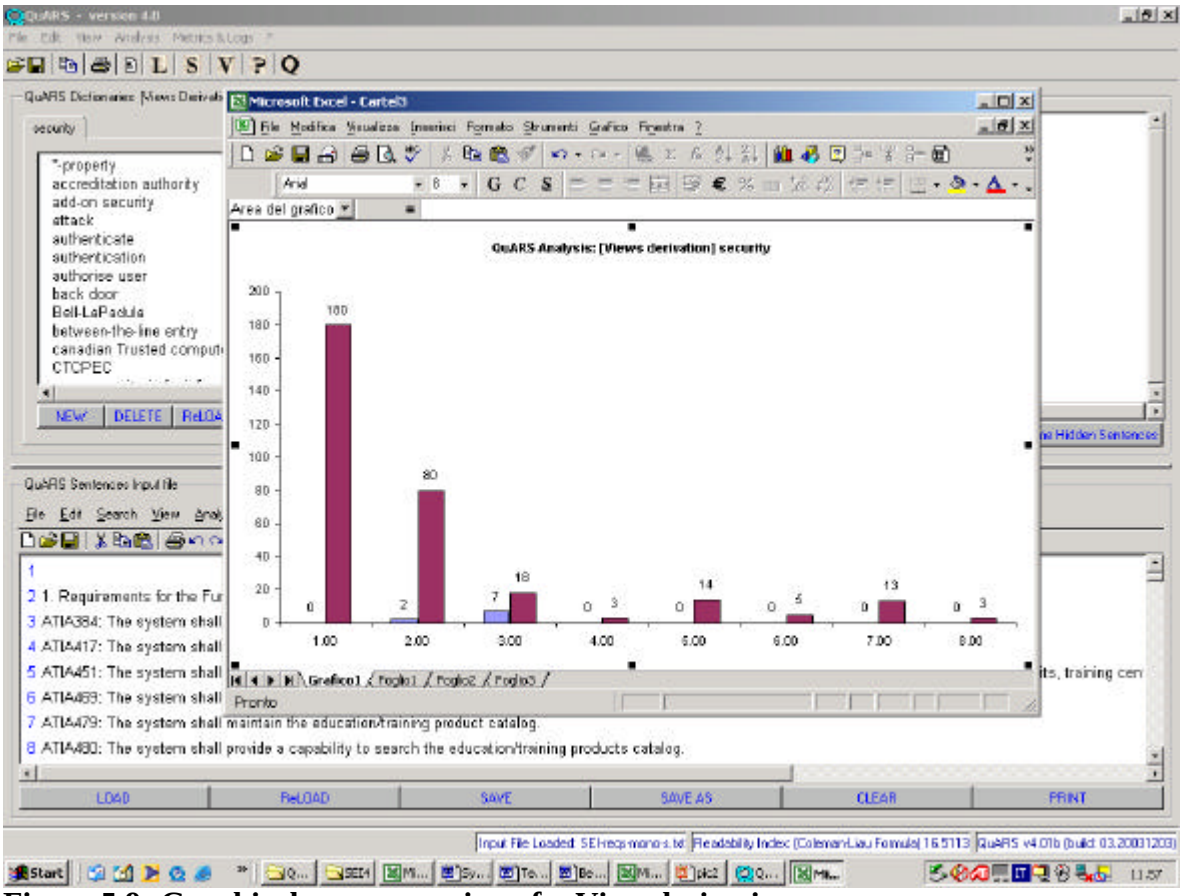


Figure 5.9: Graphical representation of a View derivation

5.4.3 Tailorability Issues

The tool QuARS has been designed to be highly tailorable according to particular application domain and different user needs. The main aspect of the QuARS tailorability is its capability to handle the dictionaries (see Figure 5.10). In fact, it possible to modify an existing dictionary and make permanent these modifications. It possible also to create new dictionaries and to delete existing ones. For this purpose a set of function buttons are available in the Dictionaries frame (Figure 5.10 – arrow 1), in the following they are described in detail:

New: this functionality allows the creation of a new dictionary. This functionality is not permitted for the syntax-based analysies because, in that case, the used dictionary is strictly connected to its analysis routine, and a new syntax-based analysis should

require a new routine The capability of adding new dictionaries for lexical-based analysis and View derivation allows the QuARS tool to be adapted to particular application domain and user needs. In fact, for a particular application domain it could be necessary, for example, do not use some terminology that in this case could be a source of ambiguity. The way a dictionary can be modified is the editing of it directly in the Dictionaries frame. In Figure 5.10 – arrow 2 the QuARS GUI in the case of creation of a new dictionary is shown.

Delete: this function allows a dictionary to be deleted. This function is not allowed for syntax-based analysies

ReLoad: This function allows to return to the last saved version of a dictionary.

Save: after some modification have been made on a dictionary, this function makes permanent the resulting new version of the dictionary.

Clear: this function cancel the content of a dictionary, this cancelation becomes permanent when the Save function is selected.

Print button: this function allows a dictionary to be printed out

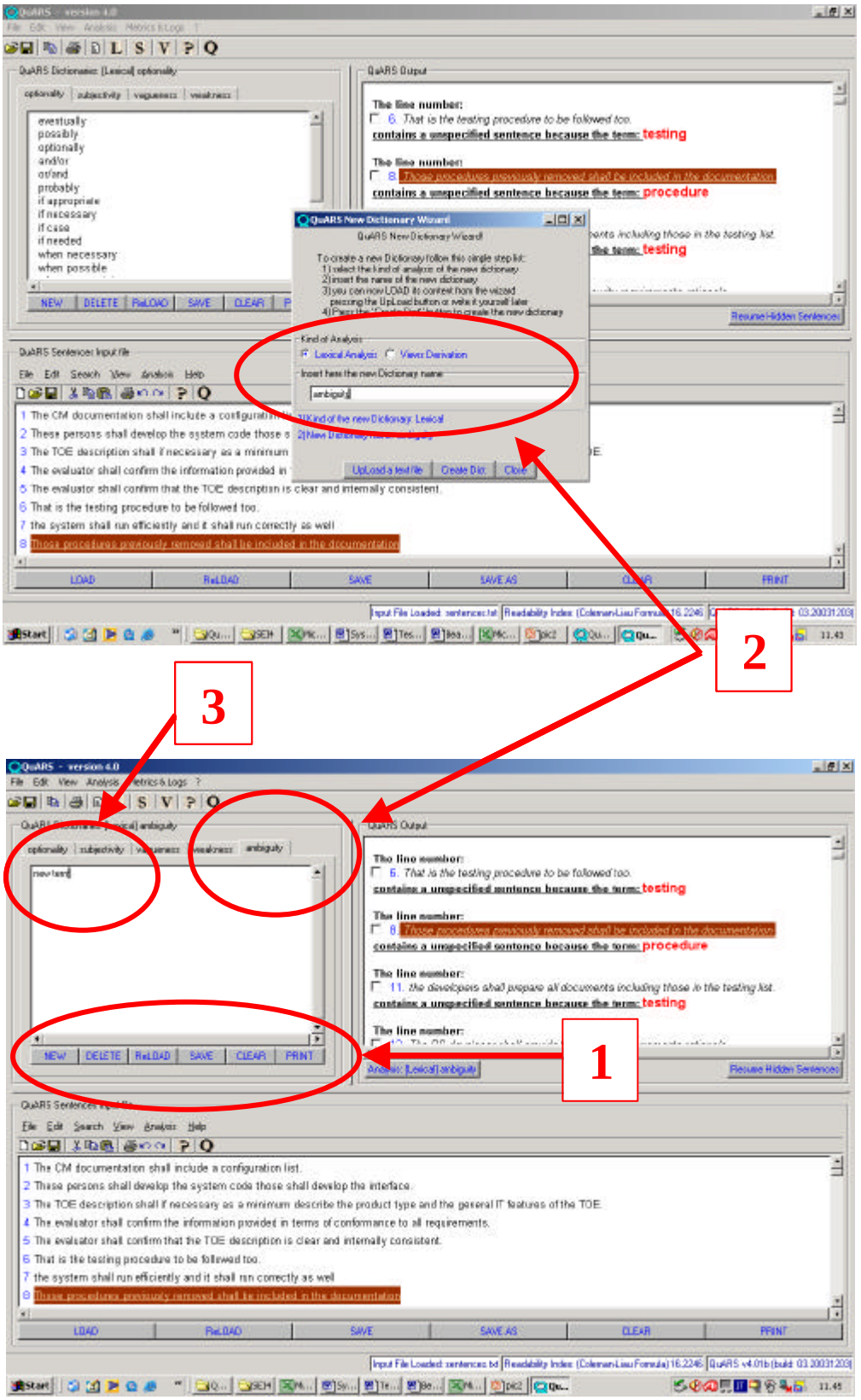


Figure 5.10: The creation of a new dictionary

5.4.4 Metrics derivation

The QuARS tool allows also the calculation of metrics about the defects rate during a requirements analysis session. Figure 5.11 shows how the GUI of QuARS allows the user to gather these metrics.

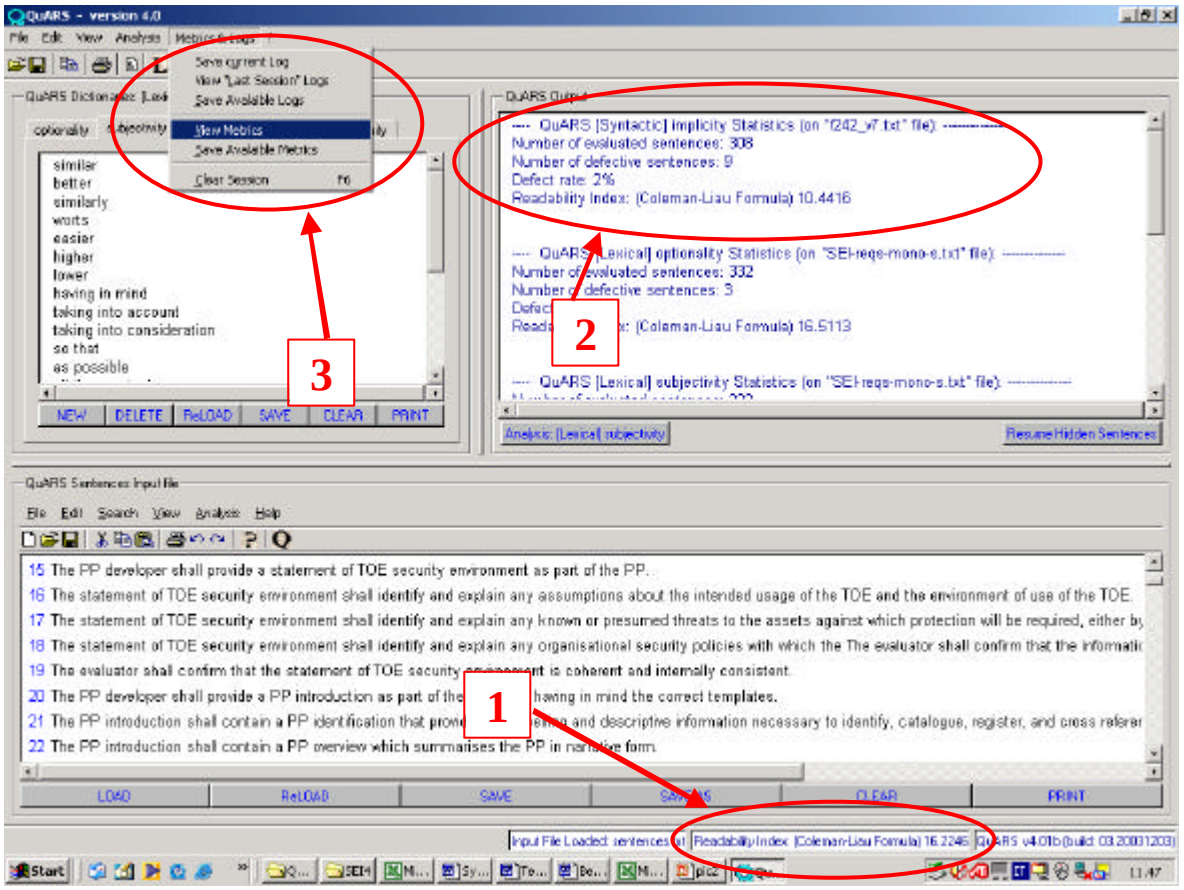


Figure 5.11: Metrics calculation

Available metrics are of two types:

Defect rate metrics: for each analysis (both lexical and syntax-based) the number of defective sentences is counted and the proportion with repect the total number of senteces in the requirements document is calculated.

Readability metrics: the Coleman-Liau Readability formula (see chapter 4 for definition) is calculated. This formula, taken from the litterature, gives a measure of

how a document is easy to be read. The formula is always available in the bottom of the QuARS GUI (Figure 11 – arrow 1).

The whole metrics collection referred to the current session can be displayed in the Output frame. For each analysis performed (even on different documents) the related metrics are maintained (Figure 11 –arrow 2). The collection of metrics can be saved as a log file by means of the special option in the Metrics&Logs menu (Figure 11 – arrow 3).

5.5 Conclusions

In this chapter an automatic tool, called QuARS, for NL requirements analysis has been presented. In the practice in the software industry the analysis of software requirements is made by humans with a clerical and borging process that consists in the reading of requirements documents looking for linguistic defects. QuARS is an innovative tool that provides the user with the capability of performing the analysis of NL requirements in an automatic way. This tool, yet in a prototypical stage, has been used for analyzing requirements documents taken from real industrial projects, in order to verify its functionalities and get a feedback form the industry on the effectiveness of the results it provides. The outcomes of these trials are encouraging because QuARS has been recognized as effective both in terms of performances and in terms of relevance of defects detected.

In particular, the tool is easy to be used and easy to be learned (because its GUI interface), it takes about 2 seconds for performing lexical-based analysis and about 15 seconds for performing syntax-based analysis of a document containing more than 400 requirements.

The tool has been designed to be highly adaptable to different application domains the requirements may belong to. Furthermore, it is able to run with almost all kind of textual requirements becuase it ask simple text files as input, and it is possible to move to a text file from almost all the formats produced by the commercial text editors.

The tool QuARS offers many improvement opportunities, in fact, the effectiveness of the analysis preformed by the tool depends on the completeness and accuracy of the dictionaries it uses. A methods, based on linguistic techniques, for enlarging the dictionaries and making them more accurate, has been defined (see section 4.4) and will be implemented as a functionality of the tool.

In order to better evaluate the impact the use of the QuARS tool on the requirements process in a industrial project, an experiment is going to be undertaken in cooperation with a software company and the Software Engineering Institute. The experiment aims at analysing with QuARS all the different versions of the requirements document of a particular project. The different versions represent the history of the revisions made on the requirements. The experiment will investigate on how many and what defects should be detected by QuARS at the first analysis session and then what is the effort that could be saved if QuARS should be used.

6. Application of the Linguistic Techniques to Use Case Analysis

The Use Case formalism is an effective way for capturing both Business Process and Functional System Requirements in a very simple and easy-to-learn way. Use Cases are mainly composed of Natural Language (NL) sentences. The use of NL as a way to specify the behavior of a system is however a critical point, due to the inherent ambiguity originating from different possible interpretations. In this chapter the use of methods based on a linguistic approach to analyze functional requirements expressed by means of textual Use Cases is discussed. The aim is to collect quality metrics and detect defects related to inherent ambiguity. In a series of preliminary experiments, a number of tools for quality evaluation of NL text to an industrial Use Cases documents are applied. The application of linguistic analysis techniques to support semantic analysis of NL expressed Use Case is also discussed. For doing that a methodology for extracting semantic infomation from a set of Use Cases has been defined and described in this chapter.

6.1 Introduction

Use Cases are a powerful tool to capture functional requirements for software systems. They allow structuring requirements according to user goals [13] and provide a means to specify the interaction between a certain software system and its environment.

Graphical object modeling languages have become very popular in recent years. Among those, UML [65] introduces a set of graphical notation elements for Use Case modeling. UML Use Case diagrams are easy to understand and constitute a good vehicle of communication. However, they mainly serve as a sort of table of content for Use Cases, presenting the connections between actors and Use Cases, and the dependencies between Use Cases.

System behavior cannot be specified in detail with Use Case diagrams. In his book [12], Alistair Cockburn presents an effective technique for specifying the interaction between a software system and its environment. The technique is based on natural language specification for scenarios and extensions. Scenarios and extensions are specified by phrases in plain English language. This makes requirements documents easy to understand and communicate even to non-technical people.

Natural language is powerful (the expression power of the English language is said to be higher than any other language in the world), well known and generally easy to understand. However, it is also prone to ambiguities, redundancies, omissions and other defects that can lead to problems when precision and clarity are essential (it is the case of software requirements specification particularly for embedded, mission-critical and performance-sensitive systems). Formal requirements specification languages (such as Z [100], B [2], LOTOS [8], etc.) were invented specifically to tackle this problem. They add formality and remove ambiguity, but are hard to understand by non-experts, which limits their practical application to some restricted domains.

6.2 Use Cases

A Use Case [12] describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment. Every Use Case constitutes a

goal-oriented set of interactions between external actors and the system under consideration. The term actor is used to describe any person or system that has a goal against the system under discussion or interacts with the system to achieve some other actor’s goal. A primary actor triggers the system behaviour in order to achieve a certain goal. A secondary actor interacts with the system but does not trigger the Use Case.

A Use Case is completed successfully when the goal that is associated to it is reached. Use Case descriptions also include possible extensions to this sequence, e.g., alternative sequences that may also satisfy the goal, as well as sequences that may lead to failure in completing the service in case of exceptional behaviour, error handling, etc. The system is treated as a "black box”: Use Cases capture who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals. A complete set of Use Cases specifies all the different ways actors can use the system, and therefore defines the whole required behaviour of the system. Generally, Use Case steps are written in an easy-to-understand, structured narrative using the vocabulary of the domain. The language used for the description is usually English. Any other natural language can be used as well, and although our analysis focuses on English, the same reasoning can be applied to other languages (considering the obvious differences in syntax and grammar rules). A scenario is an execution path of a Use Case, and represents a single path through the Use Case that leads to success in achieving the goal (the Main Success Scenario). Thus, there exists a scenario for the main flow through the Use Case, and other scenarios for each possible variation of flow through the Use Case (e.g., triggered by options, error conditions, security breaches, etc.). Scenarios may also be depicted in a graphical form using UML sequence diagrams. Table 6.1 shows the template of a typical Use Case taken from [13].

In this textual notation, the main flow is expressed, in the “Description” section, by an indexed sequence of NL sentences, describing a sequence of actions of the system. Variations are expressed (in the "Extensions" section) as alternatives to the main flow, linked by their index to the point of the main flow in which they branch as a variation.

Developers have always used scenarios in order to understand what the requirements of a system are and how a system should behave with respect to its environment. For instance, in the telecommunications domain, the use of UML sequence diagrams (formerly known as message sequence charts) is very popular. Unfortunately, this

understanding process has rarely been documented in an effective manner. The performed research is an attempt to improve the understanding process by identifying possible flaws in the textual scenario descriptions.

USE CASE #	<The name is the goal as a short active verb phrase>	
Goal in Context	<A longer statement of the goal in context if needed>	
Scope & Level	<What system is being considered black box under design> <One of Summary, Primary Task, Sub-function>	
Preconditions	<what is expected to be the state of the world>	
Success End Condition	<The state of the world upon succesful completion>	
Failed End Condition	< The state of the world if goal abandoned>	
Primary, Secondary Actors	<A role name or description for the primary actor> <Other systems relied upon to accomplish the UC	
Trigger	<The action upon the system that starts the UC>	
Description	Step	Action
	1	<the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<....>
Extensions	Step	Branching Action
	1	<Condition causing branching> <Action or name of sub-UC>
	2	

Table 6.1 Use Case template

6.3 Quality evaluation of Use Cases

In this section the application of methods and tools for the analysis of NL requirements documents in order to easily detect linguistic inaccuracies in Use Cases dealing in particular with problems related to the expressiveness of a document is discussed.

To this aim, a set of metrics that can be used to evaluate the quality of requirements documents, based on Use Cases, according to the categories listed in the previous section, has been defined. This problem has been addressed starting from the definition of a set of metrics related to quality characteristics that belong to the Expressiveness category. The metrics can be derived from the application of three different automatic tools developed to perform linguistic analysis of NL requirements documents i.e.: QuARS [23], [24], [25], [26] ARM [107], [108], SyTwo [102]. This set of metrics is based on quality properties and quality indicators used by the considered tools to evaluate NL requirements. The QuARS tool has been deeply described in chapter 5, in the following section the other two tools are described.

6.3.1 ARM

The objective of the Automated Requirement Measurement Tool (ARM) is to provide measures that can be used to assess the quality of a requirements specification document [107], [108]. ARM is not intended to be used for the evaluation of the correctness of a specified requirements document. This tool can be seen, similarly to QuARS, as an aid for "writing the requirements right," not "writing the right requirements".

In ARM, a quality model similar to that defined for QUARS is employed; this model was defined by compiling first a list of quality attributes that requirements specifications are expected to exhibit, then a list of those aspects of a requirement specification that can be objectively and quantitatively measured. The two lists were analysed to identify relationships between what can be measured and the desired quality attributes. This analysis resulted in the identification of categories of sentences and individual items (i.e. words and phrases) that are primitive indicators of the specification's quality and that can be detected and counted by using the document text file. The set of primitive indicators then has been refined by using a data base composed of words and phrases resulting from the analysis of a set of requirements specifications documents acquired from a broad cross section of NASA projects. These individual indicators have been grouped according to their indicative characteristics.

Table 6.2 shows the single sentence categories and, for each of them, the set of related indicators. The user can supply new domain-dependent quality indicators.

	CATEGORIES					
INDICATORS	IMPERATIVE	CONTINUANCE	DIRECTIVE	OPTION	WEAK PHRASES	INCOMPLETE
	shall	below:	e.g.	can	adequate	TBD
	must	as follows:	i.e.	may	as appropriate	TBS
	is required to	following:	for example	optionally	be able to	TBE
	are applicable	listed:	figure		be capable of	TBC
	are to	in particular:	table		capability to/of	not defined
	responsible for	support:	note:		easy to	not determined
	will	and			effective	but not limited to
	should	:			as required	as a minimum
					normal	
					provide for	
					timely	

Table 6.2. Standard ARM Indicators

6.3.2 SyTwo

SyTwo is a tool developed as a Web application performing linguistic analysis of an English text by means of lexical and syntactical analysis of a text. This tool can analyse the English text both to check its conformance to the rules of the Simplified English, and to detect some defects having a specific impact on the quality of

requirements. To this aim, SyTwo, which has been developed as an evolution of QuARS, partially adopts its quality model.

SyTwo builds, using a natural language grammar, the derivation trees of each sentence. During the analysis process, each syntactic node is associated with a feature structure, which specifies morpho-syntactic data of the node and application-specific data, such as errors with respect to the quality model. The output is composed of an error code, corresponding to a predefined type of defect, and of the indication of the part of the text the defects originate from.

Furthermore, SyTwo provides the value of the Coleman-Liau metrics for readability evaluation. SyTwo can point out a syntactically ambiguous sentence, when the sentence has more than one derivation tree: this implies that the sentence may be interpreted in different ways. For example the sentence "The system shall not remove faults and restore service" may be syntactically interpreted at least in these two different ways (see figure 6.1):

The negation not of the auxiliary verb shall is related to the first verb only (remove), and not to the other verb (restore). In this case, the meaning of the sentence is that the system shall not remove the faults and it shall restore the service.

The negation not of the auxiliary verb shall is related to both the verbs remove and restore. In this case, the meaning of the sentence is that the system shall not remove the fault and shall not restore the service

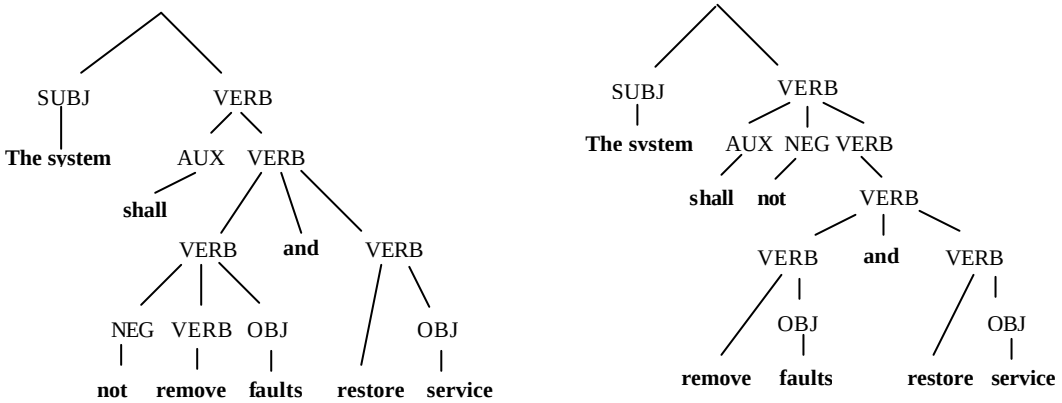


Figure 6.1: Two possible derivation trees

SyTwo is also able to capture the syntactical structure of a sentence identifying its components and their syntactic role. From this information a component of SyTwo, called Cmap, is able to extract the relations among subjects, verbs and objects in a sentence, building the so called "conceptual maps", which will be shown in section 6.6 to be useful to perform further analysis of requirements documents devoted to point out semantic problems.

6.4 Achievable Metrics

As any other evaluation process, the quality evaluation of NL software requirements has to be conducted against a model. The model is directly derivable from the Quality Models of the tools that are addressed. Starting from these Quality Models, some metrics, especially related to the Expressiveness category, can be gathered in order to perform a quantitative evaluation of a requirements document. These metrics are described in Table 3. The acronyms used in the Type column of Table 3 mean: UN = Understandability, RE = Readability, TR = Traceability, MA = Maintainability, AM = Ambiguity, SC = Specification Completion, CS = Consistency.

METRICS	TYPES	FORMULA	RATIONALE
Coleman-Liau Formula	RE	5.89*(Nl/Nw)-0.3*(Ns/(Nw/100))-15.8. Where: Nl = n. of letters in the requirements document Nw = n. of words in the requirements document Ns = n. of requirement sentences in the requirements document	It measures the difficulty in reading the document
Average number of words per sentence	RE, UN	Nw / Ns	Short sentences make the requirements document more readable/understandable
Continuance Index	TR, MA	Ncon/Ns. Where: Ncon= n. of conti-nuances in sentences. Continuances are phrases	The use of continuances indicates a well structured document, but too many

		as “the following:” that follow an imperative verb and precede the definition of lower level requirement specification (see Table 2)	continuances indicate multiple, complex requirements
Comment Frequency	UN	Nc / Ns. Where: Nc = n. of comment sentences.	The comments within the requirements document reduce the risk of misinterpretations
Directives Frequency	UN	Nd / Ns. Where: Nd = n. of directives (see Table 2). Directives are words or phrases that indicate examples or other illustrative information	Directives make the document more understandable.
Multiplicity	UN	Nmul / Ns. Where: Nmul = n. of sentences having more than one main verb or more than one direct or indirect complement that specifies its subject.	The presence of multiple sentences makes the requirements document more difficult to be read and understood
Vagueness	AM	NVag / Ns. Where: NVag = n. of sentences including words holding inherent vagueness, i.e. words having a non uniquely quantifiable meaning.	The presence of vague sentences increases the level of ambiguity of the requirements document
Subjectivity	AM	Nsub / Ns. Where: Nsub = n. of sentences referrings to personal opinions or feelings.	The presence of subjective sentences increases the level of ambiguity of the document
Optionality	AM	Nopt / Ns. Where: Nopt = n. of sentences containing an optional part	The presence of optional sentences increases the level of ambiguity of the document
Weakness	AM	Nwea / Ns. Where: Nwea = n. of sentences containing a weak main verb.	The presence of weak sentences increases the level of ambiguity of the requirements document

			requirements document
Underspecification	SC	Nusp / Ns. Where: Nusp = n. of sentences having the subject containing a word identifying a class of objects without a specifier of this class.	The presence of underspecification makes the requirements document not fully specified
Implicity	UN	Nimp / Ns. Where: Nimp = n. of sentences having the subject generic rather than specific.	The presence of implicit sentences makes the requirements document prone to be misunderstood
Under-reference	CO	Nure / Ns. Where: Nure = n. of sentences containing explicit references to: -unidentified sentences of the requirements document itself; - documents not referenced into the requirements document itself - entities not defined nor described into the requirements document itself.	The presence of these references introduces inconsistencies in the requirements document
Unexplaination	UN	Nune / Ns. Where: Nune = n. of sentences containing acronyms not explicitly and completely explained within the requirements document itself.	The presence of acronyms which are not explicitly and not completely explained makes the document prone to be misunderstood

Table 6.3 Achievable metrics

6.5 A Case Study

As a case study, a requirements document, taken from an industrial project has been considered. The document has been analyzed with QuARS, ARM and SyTwo. This document, provided by Nokia, describes the functional requirements for the user

interface of a new feature (FM radio player) to be included in a line of mobile terminals. This feature was meant to provide the possibility to use a phone as a built-in stereo frequency modulation (FM) radio. The first product to include this feature has been the Nokia Mobile Phone model 8310.

The document analysed is composed of about one hundred Use Cases. The outcomes in terms of the proposed metrics are reported in Table 6.4. The information about the quality of the analysed document provided by these metrics may be summarized as follows.

Observing the values obtained from the calculation of the metrics 1, 5 and 7, it can be observed that the terms used in the requirements were not properly selected. In the following some samples of defective sentences related to these metrics taken from the analysed Use Cases are proveded:

This procedure is performed by the user to enter the frequency (Implicit sentence: indicator this).

In addition, the user is naturally able to adjust the volume (Vague sentence: indicator naturally)

The user can switch the radio on by selecting `Radio` from the menu (Under-specified sentence: indicator menu).

The word "menu" has been set as under-specified by the tool users. However, while generally the sentence must be recognized as under-specified, and it is good to have its under-specification pointed out by the tool, in this particular case the detection of the defect may not trigger any improvement actions on the document. This is because the user interface configuration and styling is done independently of (and after) component development and integration. Therefore, it may be a methodological choice to leave this defect unsolved until the very end of the software integration phase.

	Metrics Name	Reference values	Actual value	Used tool
1	Vagueness	The closer it is to 0 the more unambiguous the requirements document is	4	QuARS/ SyTwo / ARM2.1

		requirements document is		
2	Subjectivity	The closer it is to 0 the more unambiguous the requirements document is	0	QuARS/ SyTwo
3	Optionality	The closer it is to 0 the more unambiguous the requirements document is	0	QuARS
4	Weakness	The closer it is to 0 the more unambiguous the requirements document is	0	QuARS/ SyTwo/ ARM2.1
5	Under-specification	The closer it is to 0 the better specified the requirements document is	19	QuARS
6	Under-reference	The closer it is to 0 the more consistent the requirements document is	0	QuARS
7	Implicity	The closer it is to 0 the more understandable the requirements document is	12	QuARS
8	Unexplaination	The closer it is to 0 the more understandable the requirements document is	0	QuARS
9	Coleman-Liau Formula	Typically ranged from 0,4 (easy) to 16,3 (difficult)	17.6	SyTwo
10	Average number of words per sentence	Simple sentences have a number of words less than 10 – 12	14,82	QuARS
11	Continuance Index	Optimal range: 0.1 – 0.2	0	ARM 2.1
12	Comment Frequency	Optimal range: 0.1 – 0.3	0,04	QuARS
13	Directives Frequency	Optimal range: 0.1 – 0.3	0,08	ARM 2.1
14	Multiplicity	The closer it is to 0 the more understandable the requirements document is	12	QuARS

Table 6.4 Metrics values

The values of metrics 9, 10 and 14 indicate that the sentences of the document need to be simplified in order to decrease the risk to be misinterpreted. Below a sample of a multiple sentence taken again from the analysed document:

The phone displays the confirmation note Frequency set and goes to the FM Radio state displaying the selected frequency with the channel number and name if a channel in that frequency has already been saved earlier.

To avoid the problems associated with the multiplicity this sentence should be split in more than one simpler sentence.

The values of metrics 12 and 13 seem to indicate that the document is poor of extra information that might make it more understandable. However, the reference values for these two metrics are derived from the good practices of NL requirements, and they could be not fully significant for Use Case requirements, because these kinds of requirements specifications are inherently more descriptive.

6.6 A Relation-based Appraoch for the Analysis of Use Cases

In this section it is discussed how the application of NL based techniques can provide an effective support to deal with Consistency and Completeness issues of requirements expressed by means of Use Cases.

To effectively address the Consistency and Completeness aspects of requirements specifications, we should resort to their formalization [36], [110]. Indeed formal methods are a powerful mean to evaluate requirements since they provide a theoretical framework in which their correctness can be verified. Formal methods require, however, a specific skill and this increases their application cost preventing their wide application in industries. Here it has been followed a light-weight application of formal reasoning by means of a study on the relations between actors, with the purpose of facing consistency and completeness problems in the requirements documents.

We can observe that a system specification written as Use Cases is structured in three semantic layers:

1) the specification is, at its higher level, composed of a set of Use Cases plus other artefacts and models; each Use Case defines a goal for a primary actor and some secondary actors, establishing relations among actors.

2) in each Use Case the scenario and its extensions play a major role in specifying the system behaviour; that is they define the sequential control flow, with exceptions defined by the extensions.

3) each scenario or extension sentence has its internal, linguistic structure, which defines a relation among (primary and secondary) actors and the operations they perform or take part into.

It is on the third layer that the linguistic analysis has an immediate application, but the structure of the previous layers gives important information as well. Our aim is the definition of a relational structure combining both the results of the linguistic analysis on such sentences and the structure implied by the other layers.

The methods under investigation strictly rely on the structure of the Use Cases and are based on the "functional" relations, i.e. the relations or dependencies between actors of a Use Case-based description of a system.

The methods and tools presented in this chapter also rely on the structure of the Use Cases and are based on the study of the relations between actors of Use Case-based description of a system. The method described in this chapter can be placed between a "lightweight" parsing [84] and a "full-fledged NL" approach [81] and aims at demonstrating that the extraction of "semantic" information for a text is possible also without using tools and methods too heavy.

The relations of interest are the "functional" relations, i.e. the relations or dependencies between two actors. These relations can be determined looking at the syntactical structure of each sentence of the Use Case scenarios defining a set of items (quadruples) where each primary actor (the subject of the sentence) has been put in relation with the secondary actor (the complement) according to the verb. The canonical form of these relations is:

(1.) (Actor_1, verb_i, Actor_2, Use_Case_id).

Each item compliant with (1.) describes an occurrence of a functional relation between two actors established by the verb and indicates the Use Case in which this relation occurs.

The functional relations between two actors, in the form (1.), can be extended, by transitivity, to other actors when two items with the following form exist: (A_i, v_1, A_j, UC_x) and (A_j, v_2, A_k, UC_y). In this way, hence, an indirect functional relation between the actor A and the actor A_k is also established by transitivity. Starting from this consideration, chains joining different actors can be built, where each item (A_i, v_x, A_j ,UC_x) of the chain is such that the previous item has the form (A_k, v_y, A_i, UC_y) and the following has the form (A_j, v_z, A_h, UC_z).

The collection of all the items derivable from a Use Case based requirements document is said Relations core. The relations between actors that can be extracted directly by the NL description.

We can derive specific, non-elementary, relations from the relations core. In the following some definitions and define some properties based on the elementary relations (1.) are provided.

The ignores relation, denoted by A~B, holds if no relation (A, verb_i, B, Use_Case_id) exists.

The relation (1.) between actors can be used to build the Relation Graph. The nodes of this graph represent the actors and an oriented arc connecting two nodes (A and B) indicates that A drives B. Two nodes are adjacent if an arc from A to B exists. A path from the node A to the node B on this graph is a sequence of adjacent nodes in a graph starting from the node A and arriving to the node B. On the basis of the Relations graph some further relations between actors can be defined:

The is connected to relation, denoted by A => B , holds if at least one path from A to B exists on the graph.

The is chief relation, denoted by A =>> B, holds if B ignores A and A is connected to B.

The chief graph (derived from the chief relation) is an acyclic graph composed of nodes (the actors) and oriented arcs connecting two actors, an arc originating from the node A and arriving in the node B means that A is chief of B.

Nodes of the chief graph having no incoming arcs are said leader nodes and nodes having no out arc are said executors nodes. An example of Relation and Chief graphs are provided in Figure 6.2.

The availability of the functional relations and of the graphs derived from them enables the capturing of some semantic information on the system to be described. In particular, this information can be used to support the detection of critical points (in terms of consistency and completeness) in the interactions between different actors. These critical points can be revealed by analysing the set of derived direct and indirect relations.

The derivation of the relations core and the consequent construction of the relation chains, relations graph and chief graph, can be supported by automatic tools based on NL processing techniques.

In fact, the basic relations (1.) are detectable by using a syntactical parser able to identify the different components of a NL sentence. To our purpose, the key components to be identified are the subject(s), the verb and the complement(s) associated to the verb. Once this information is achieved, it is possible to define the relations and to build a data base containing the relations core derivable from the collection of Use Cases under analysis.

6.6.1 An example of Derivation of Relations

In this section, an application of the relational approach to a sample Use Case document is presented, with the aim to clarify the concepts discussed above. The example presented in this section is derived, with few changes, from a sample System Requirements Document available on the web at the Cockburn's book [14], which is provided in Appendix 1. This document, describing a Purchase Request Tracking System, has the purpose to provide the functional requirements of a basic system for the official Buyers of the Cmpany, to track what they have ordered from Vendors against what they have been delivered. The documents is organised as a set of Use Cases.

The primary actors of this document are:

- Approver: typically the requestor's manager, who must approve the request.

- Authorizer: person who validates the signature from the Vendor.

- Buyer: person who manages the order, talking with the Vendor.

- Vendor: person or company who sells and delivers goods.

- Requestor: person putting in a request to buy something.

- Receiver: takes care of the arriving deliveries

The document contains fifteen Use Cases describing the behaviour of the system. It has been slightly modified by adding two new Use Cases to make it more precise and suitable for the analysis. The Use Cases included into the document are compliant with the Cockburn's style, and they include several data such as, for example, Preconditions, Postconditions, Trigger, Extensions, etc. Each Use Case has been simplified by reducing the information associated to them. In particular, only the Primary Actor, the statement of the Goal and the description of the Scenario have been taken into account.

These data represent the minimum set of information necessary to save the essential meaning of the Use Case. In Appendix A, the set of the simplified Use Case used for the experiment is shown.

The outcomes of the application of the relational approach to the simplified Use Cases of the case study are summarized in a collection of relations items between actors and a set of relations chains derived from the relations items.

For simplicity let us identify the actors of the case study by a letter:

A. Authorizer
B. Approver
C. Requestor

D. Buyer
E. Vendor
F. Receiver

Figure 2. contains the relations derived from the case study where each actor is identified by the corresponding letter along with the corresponding relation graph and chief graph.

In the following a possible set of relations chains starting from the relation (A, notify, B, UC3), is provided:

- (A, notify, B, UC3), (B, send, C, UC6), (C, send, B, UC7).

- (A, notify, B, UC3), (B, send, C, UC6), (C, send, B, UC8).

- (A, notify, B, UC3), (B, send, C, UC9), (C, send, B, UC7).

- (A, notify, B, UC3), (B, send, C, UC9), (C, send, B, UC8).

- (A, notify, B, UC3), (B, send, A, UC12).

- (A, notify, B, UC3), (B, send, A, UC12), (A, change, B, UC3).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, C, UC9).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, D, UC16).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, C, UC9), (C, send, B, UC7).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, C, UC9), (C, send, B, UC8).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, D, UC16), (D, change, E, UC4).

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, D, UC16), (D, send, C, UC9).

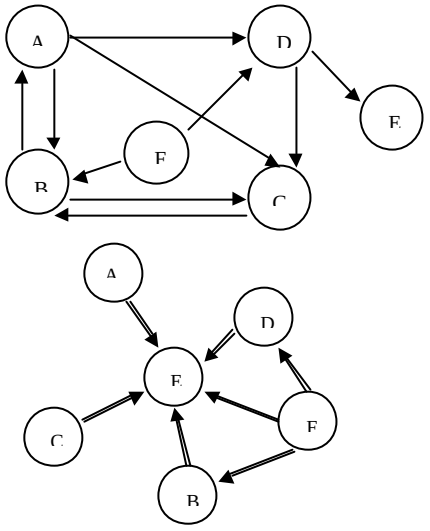
- (A, notify, B, UC3), (B, send, A, UC12), (A, send, D, UC16), (D, send, C, UC9), (C, send, B, UC7)

- (A, notify, B, UC3), (B, send, A, UC12), (A, send, D, UC16), (D, send, C, UC9), (C, send, B, UC8).

The table in figure 6.2 also shows that an ignore relation between C and A occurs, and it means that C doesn't influence directly the A's behaviour.

It is to be noted that the tool QuARS (see chapter 5) can be used to derive the basic relations and then the relation chains and the different graphs can be built from them.

Primay Actor	verb	Secondary Actor	UC
A	change	B	3
A	notify	B	3
A	notify	B	3
D	change	E	4
C	send	B	5
B	send	C	6
C	send	B	7
C	send	B	8
B	send	C	9
A	send	C	9
D	send	C	9
D	return	E	11
B	send	A	12
D	send	E	14
F	notify	D	15
A	send	D	16
F	send	B	17



Relation Graph

Chief Graph

Figure 6.2 Example of relations between actors and related graphs

6.7 Applying the Relational Approach

In this section possible applications and developments of the relational approach to the Use Casebased requirements engineering is discussed.

Since relations indicate the presence of a verb in the Use Case relating two actors, they often indicate possible interactions between actors. Hence, relations chains can be interpreted as interaction schemata. Walkthroughs of these interaction schemata

may be performed in search of undesired, inconsistent and incomplete dynamic behavior of the system.

These schemata may also form the basis for a formal analysis of interactions, which, however, are not addressed in this context.

Walkthroughs of interaction schemata may be aimed at detecting relation chains containing loops, because loops indicate a more complex kind of interaction, and may point to a possible synchronization problem (such as a deadlock). It is possible, in this way, to point out some potential synchronization problems in a sequence of actions.

Let us consider, for instance, the example of section 4.2. In this case, the relation chain: (A, notify, B, UC3),(B, send, C, UC6), (C, send, B, UC8), presenting a loop can be detected. If we carefully walk through this chain, and we represent this interactions sequence on a time scale (see figure 6.3), it is possible to understand that some potential synchronization problems may occur.

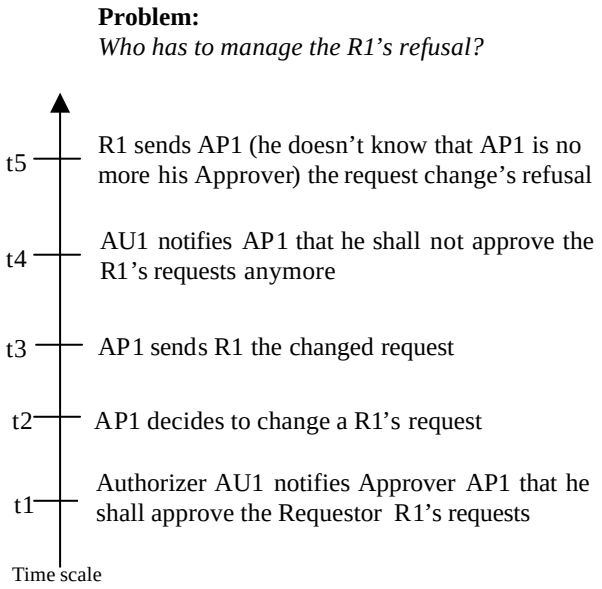


Figure 6.3 Interactions sequence

In fact, if the Approver B1 sends the changed request to the Requestor C1 and, before that C1 tells B1 that this change is refused, Authorizer A1 changes the authorization to B1 and makes B2 the Approver of C1, then who should manage C1's refusal?

In this case, it is possible to detect an inconsistency in the requirements due to an incomplete specification of the requirements because the notification of the changed Approver is not sent also to the associated Requestors. This kind of problems, that are hidden if we consider only the Use Cases-based requirements document, may be easily detected by using the relations chains.

Another possibility of exploiting this information is to point out those pairs of actors that have an higher number of interactions than the others. The pairs that, in the case study, have the highest number of different interactions are Approver-Requestor (5 interactions), Authorizer-Approver (4 interactions) and Buyer-Requestor (3 interactions). The indication that can be derived from these data is that the interactions between these actors are at the core of the functionality of the system, and therefore should be analysed in more detail in order to point at possible problems. Also, this information can give an indication to which parts of the system should be stressed at testing time.

The semantic information that can be extracted from the derived relations and graphs can help the analysis of correctness and completeness of the requirements by detecting some gaps in the specification of Use Cases.

In fact, the graphs defined above (and in particular the chief graph) allow some interesting considerations to be made. The chief relation is not to be intended as determining a hierarchy in terms of the importance of the role played by the actors. This relation and the information derivable from the graph is a semantic information that allows to enlighten the influence of an actor on the others.

In particular, if a node A of the chief graph is connected with the node B by an arc (A, B), then it can be argued that the behaviour of B doesn't influence that of A. This kind of semantic information about the actors, that cannot be directly derived from the set of Use Cases, can play a relevant role for the analysis. In particular, it is possible to easily detect lacks in the relational structure of the requirements.

In the example shown above, the relation F =>> D occurs. This occurrence enlightens a gap in the specifications because the buyer should have the capability of having a relation with the receiver (for instance, to ask the status of an on-going acquisition).

The relational approach can be oriented to achieve a guidance for systematic construction of the Use Case requirements documents. In fact, building the relations graphs in parallel with the definition of the Use Cases impels a continuing series of walkthroughs to check the part of the relations graph completed so far and examine how remaining relations should be added to the graphs themselves.

We wish, in the end to point at another application of the relational approach, which spans outside the context of Use Cases. A concept that has gained importance in the last years, especially in the telecommunication field, is the concept of feature. A feature is a capability of a system which provides value to the users, but is conceived as separate from the other features provided by a system to its users. However, at the system level, features can interact in a complex manner (a problem often referred as "Feature interaction"), so they cannot be treated as separate in the development of the system, and especially in the requirements document. A feature may even prevents other system activities: for instance, in a mobile handset user interface the "keyguard" feature prevents almost all other user-originated activities (but not incoming call handling).

The description of a feature by Use Cases can be trivial (in the keyguard example the scenario might be composed simply of the "set the keyguard on" activity) and the Use Cases may be not able to represent how the system behaviour is affected by a feature. The knowledge of the influence of the features on the UCs can be important mainly for the testing of the system because the Use Cases are not enough for representing the consequences of the features on the functionalities they describe.

For this reason the relational approach to the Use Case analysis can be of interest to identify those Use Cases affected by a feature. For example the Use Cases affected by the keyguard feature can be detected because they have in their scenario a sentence like "User digit a key". These UC are influenced in case of the keyguard is set on.

6.9 Conclusions and Future Works

Use Cases allow functional requirements to be captured in an effective way by means of scenarios. Developers have always used typical scenarios (often in graphical form) in order to understand what the requirements of a system are and how a system

works; Use Cases provide a means to rigorously express requirements along these lines.

In this chapter an approach to the analysis of Use Case requirements documents based on the relations between the actors is presented. Starting from the simple relation between two actors derivable from a scenario sentence, by means of NL parsing tools, some more complex, derived relations have been defined. These relations are able to provide semantic information on the content of a requirements document, supporting the completeness and consistency analysis. The semantic information on the Use Case requirements documents that can be captured with this approach is only partial, w.r.t. the semantic of the whole requirements. Anyway, this information is able to provide a concrete support for the analysis. The use of the semantic information derivable with the relation-based approach has been discussed in this chapter. In particular, the knowledge of the functional relations between actors expressed by the Use Cases allows to perform walkthroughs in the relation core to detect possible gaps in terms of consistency and completeness. Moreover, a guidance for a systematic construction of the Use Cases requirements document can be obtained by the parallel development of graphs and schemata representing the relations.

A related work to ours is that reported in [94], in which more sophisticated NL techniques are used to extract concept lattices out of Use Cases, which offer a richer information to the analysis. Our approach use simpler, low cost NL techniques to extract useful information: it would be interesting to see whether the benefits obtained by heavier NL techniques balance their higher costs.

The relation-based approach to the analysis of Use Cases is a promising research direction because it can be used as a mean to bridge the gap between the use of the informal NL descriptions typical of requirements documents, and the more formal artefacts typical of later stages of the development process. In particular, the study of the relations between actors, though starting from a light formalism as the Use Cases are, can provide enough information to move towards the application of formal methods with the support of automatic tools and in a user friendly way. It is planned to investigate at this regard the annotation of the relation graph with pre-conditions and post-condition in order to perform simulations of the system and perform a more refined analysis.

Another subject that is under investigation is the extraction of test cases from Use Case scenarios. Also in this case, extracting information from the textual descriptions in the form of relations between actors helps in the definition of test cases covering the most intricate interaction schemes.

7. Representation and Verification of Use Cases for Product Lines

Capturing the variations characterizing the set of products belonging to a product line is a key issue for the requirements engineering of this development philosophy. This chapter describes a way to extend the well-known Use Case formalism in order to make possible the representation of these variations. The proposed formalism allows the representation of the constraints the products beloning to a product line shall respect and provides a way to verify the conformance of a set of related Use Case. This paradigm has been defined in the perspective to make them suitable for an automatic representation and verification.

7.1 Introduction

The Product lines and Use Cases are two important and well-established paradigms of modern industrial development. [2,8,10] This introduction briefly describes them, underlining the reasons why they are becoming so popular.

The need for quality, easy reuse, minimization of costs and times of development of new products lead to the adoption of the Product Line (also known as Product Family) approach. A product line can be seen as a set of products with common characteristics

which link them together. While developing a product line it is possible to move from the family level (which represents those common features) to the product level (which represents the single product, with all its particular characteristics) by an instantiation process, and on the contrary from the product level to the family level by an abstraction process.

One of the main reasons to use a Product Line approach is reuse, which extends far beyond mere code reuse. Each single product can be developed following the analysis, design, coding, planning and testing efforts already done for previous products of the same Product Line. The advantages of reuse come however at some cost:

The architecture of the product line provides a template for every single product of the family which will be developed: this means that investing a good amount of energy designing a solid yet flexible architecture will lead to a simpler and less error-prone development of the company's products. However, this also means that the architecture needs to be open to deal with issues such as variabilities [7] which determine additional constraints, costs and efforts.

There is a need of tools and processes to help managing variation and making changes to the products: these kinds of tools and processes need to be more solid than those used for single products, thus they are more expensive and complex. However, the tools can be used for every product of the family, and the initial complexity later favours an easy development and reuse.

Common software components can and must be developed with higher level of quality, because they are used in every single product. This implies a reduction of costs and time for the development of many products, but it also means that common components must be robust and applicable across a wide range of product contexts, thus raising their complexity and development costs.

Workers play a role in many products at a time instead that in only one. This enhances personnel mobility among different projects and rises productivity, but in order to reach these advantages, training is needed, which implies initial additional costs.

Due to the initial costs needed to adopt a Product Line approach, some companies have been reluctant. However, it was widely documented how the advantages of using Product lines largely overcome the disadvantages and the initial effort needed to

change the organization of the work inside a company. Also, a reactive, more relaxed product line approach can be used for those companies which cannot afford the risks and costs of a more proactive approach.

Use cases [3] are an easy, natural way to express functional requirements of a system. Their popularity derives from the simplicity of their approach: a well structured, easy to understand document written in controlled natural language.

Use cases are widely used in modern industrial development, so it seems natural to try to find an effective way to combine them with the Product Line paradigm. While use cases are already used in this context, the real challenge is to be able to semi-automatize them in order to instantiate single products Use Cases from more general ones.

We have proposed the notation of Product Line Use Cases (PLUC) [1], [6], a version of the notation of Cockburn’s use cases [4] aimed at express requirements of product lines. Cockburn’s use cases allow the functional requirements of a system to be described, by imposing a specific structure on requirements documents, which separates the various cases in which the system can be used by external actors, and for each case defines scenarios of correct and incorrect usage. The PLUC notation is based on structuring the use cases as having two levels: the product line level and the product level. In this way product-related use cases should be derived from the product line-related use cases by an instantiation process.

In this chapter it described how on this notation has been eleborated by adding the possibility of expressing constraints over the product-related use cases that can be derived from a product line use case. The constraints are expressed as Boolean conditions associated to the variability tags. Using this notation, it is possible to express in the requirements document of the product line not only the possible variant characteristics that can differentiate products of the same family, but also which combinations of variant characteristics are “legal” and which are not.

This approach is based on the proposal by Mannion [11] that addresses general product line model requirements: he presents a way to describe the relationships between product line requirements, in order to formally analyze them and to extract information about the internal consistency of the requirements (i.e.: they provide a valid template for at least one single product) and of the single products derived from the product line model (i.e.: they satify all product line requirements' constraints).

A similar approach has been adopted and it has been applied to the PLUCs, by transforming the described relationships between PL requirements into relationships between PLUC tags and between different PLUCs, and the set of basic relationships with some composed new ones has been extended.

The information added to PLUC provides on one hand the ability of automatically checking whether a product-related use case is conformant to the family requirements; on the other hand, the adoption of constraint-solving techniques may even allow for automatic generation of product-specific use cases from the family level use cases document..

7.2 PLUC Notation

Use cases are a way to express functional requirements of a system. A use case defines a goal-oriented set of interactions between external actors and the system under consideration. Actors are parties outside the system that interact with the system. An actor may be a class of users, roles users can play, or other systems. There are two kinds of actors: primary actors and secondary actors.

A primary actor is one having a goal requiring the assistance of the system.

A secondary actor is one from which the system needs assistance.

A use case is initiated by a primary actor to achieve a goal, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to accomplish the task which will lead to the goal. It also includes possible alternative sequences which can arise due to errors, alternative paths, etc. The system is often treated as a black box.

In [1] the classical use case definition given by Cockburn in [4] to product lines has been extended, adding variability to this formalism. The result are Product Line Use Cases (PLUC**), which are essentially use case which allow variability through the use of special *tags*, in order to derive single Product Use Cases (**PUC**).**

In PLUCs variations are implicitly enclosed into the components of the use cases. The variations are then represented by tags that indicate those parts of the product line requirements that need to be instantiated for a specific product in a product-related document. In figure 2 a UML description of a PLUC is provided.

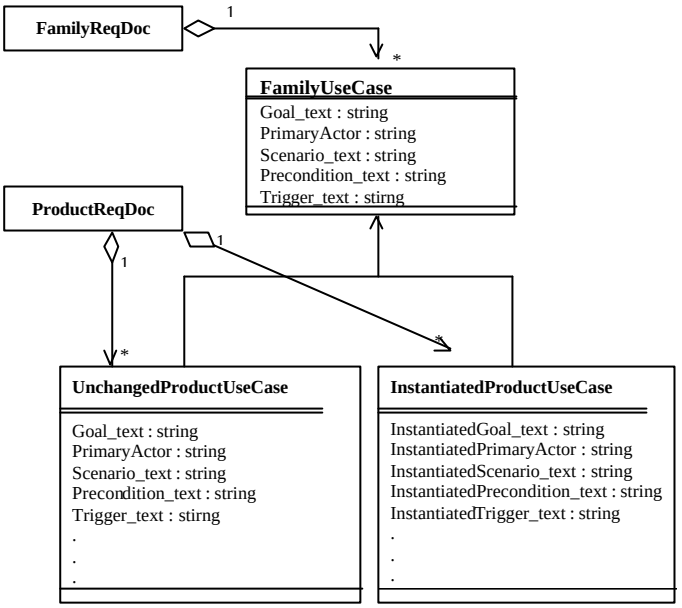


Figure 7.1: UML description of PLUC

Product line requirements can be considered, in general, as composed of a constant and a variable part [1, 9]. The constant part includes all those requirements dealing with features or functions common to all the products in the product line and, for this reason, do not need to be modified. The variable part represents those aspects that can be changed to differentiate a product from another. A possible extension of use cases to express variability during requirements engineering of Product Lines is based on structuring the use cases as having two levels: the product line level and the product level. In this way, use cases for a specific product are derived from the generic use cases by an instantiation process.

For the generic use cases, the variations are represented by tags that indicate those parts of the product line requirements that need to be instantiated for a specific product in a product-specific document. For doing that, tags are included into the use case scenarios (both main scenario and extensions) in order to identify and specify variations. The tags can be of three kinds:

Alternative: they express the possibility to instantiate the requirement by selecting an instance among a predefined set of possible choices, each of them depending on the occurrence of a condition;

Parametric: their instantiation is connected to the actual value of a parameter in the requirements for the specific product;

Optional: their instantiation can be done by selecting indifferently among a set of values, which are optional features for a derived product.

The instantiation of these types of variabilities will lead to a set of different product-related use cases.

This extension of the use cases representation is called PLUC, and two examples are provided in Figure 3.The example in Figure 3 describes the behaviour of the phones belonging to the family when a game is played by the user. In both the examples the variation points are represented by means of tags according to the PLUC formalism.

A PLUC describes the general behaviour which all family members (PUCs) should have during the accomplishment of a specific task: it acts like a template from which is possible to derive single PUCs by the instantiation process of its tags, which can be of many different types, as it will be detailed in the section 7.3.

7.3 PUC derivation from PLUC

In this section the approach to formalize variabilities for specifying the PLUC and a way to effectively verify the compliance of a PUC to the family constraints are described.

7.3.1 Specification of the PLUC

The specification of the tags into the PLUC is a critical step for making the PLUC approach effective in practice. The definition of a method to formalize the three kinds of tags described in Section 2 (Alternative, Optional, and Parametric) is a necessary preliminary step for the verification of the compliance of a PUC to the family

constraints. In fact, the constraints that characterize the products belonging to a family can be expressed in terms of the relations among the different tags indicating the variation points in a PLUC.

To express the variability tags of the PLUCs in a formal way we have to take into account all the possible situations which can arise during the writing of a PLUC, paying particular attention to the variable tags of the PLUC itself.

First of all, we have to define the formalism to be used for expressing those relationships: propositional calculus is a simple and effective way to describe them at high level, so we will use propositional connectives between PLUCs components. According to this formalism the basic symbols used in the following formulas are '||' (the logical OR operator), '&&' (the logical AND operator) , '==' (the 'equal to' logical operator) and '~' (the logical NOT operator). The operands of the expressions representing the different tags included in a PLUC are the variabilities to be instantiated when moving to a PUC.

A formalism to describe the essential set of tags is described in the following. For each type of tag a logical expression able to capture its meaning is described:

The *alternative tag* indicates mutual exclusion, which means that during the instantiation process one and only one from a set of different values can be assigned to the tag. This type of relationship can be expressed with a logical Exclusive or**.**

The *optional tag* represents a subset of a PLUC steps that can or cannot be present in an instantiated PUC, depending of the value of some other instantiated tag (i.e. if a mobile phone type contains game C, the PUC called "starting a game" will have a step "print GAME C on screen", otherwise this step will not be present in the PUC). The correct propositional connective to be used for this type of relationship is Bi-conditional:

a bi-cond b iff

[(a==true)&&(b==true)]||[(a==false)&& (b==false)].

The *parametric tag* indicates that some subsets of PLUCs steps can be chosen in a way that at least one of them will be chosen to be inserted in a specific PUC, but more than

one is allowed to be chosen (i.e. there can be more than a way to start a game in a mobile phone interface, and at least one must be present). This relationship is modelled with a logical or**.**

It is possible to define some more complex and structured relationships, which can be used to more easily describe some common situations that can be found when a PLUC is read through.

As we can have tagged steps which have to be present if another tag has a particular value, we can also have tagged steps which have not to be present if another tag assumes a particular value. This is simply the opposite of the logical Bi-conditional**, it is called logical** Excludes **and it is a mean to include logical** not **into the set of logical predicates: given two tagged steps a and b, the following relationship can be established:**

a excludes b iff not[a and b] or [(not a) and (not b)]

Sometimes we can choose zero or more steps from a subset of PLUC steps. This situation is modelled by the use of logical Bi-conditional **and logical** or **at the same time: given a tagged step a and a set of tagged steps (b_1, … , b_n), we can establish the following relationship:**

a excludes (b_1, … , b_n) iff

`[a and b_1] or [(not a) and (not b_1)] or … or [a and b_n] or [(not a) and (not b_n)]`.

It is possible to define other new logical relationships, simply using the basic ones presented.

The constraints that define the borders and the characteristics of a family and that must drive the specification of a PUC are expressed by means of the formalization of the tags as seen above. These tags may be considered as the way to represent the conditions to be satisfied in order to make a variability solution not contradictory with the family characteristics.

7.3.2 Derivation and Verification of a PUC

In this section the instantiation of the PLUC tags to derive a PUC and the method to be used for the verification of the compliance of the PUC to the family constraints set up in the tag description are discussed.

The process of instantiating tags consists of assigning an actual value to each variable appearing in the tag expressions of PLUCs of interest. The instantiation of the tags expressing the variabilities of the family corresponds to the definition of the compulsory characteristics of the PUC we are deriving. In other words, the instantiation of the tags defines the requirements of a particular product belonging to the family.

A PLUC consists in a series of steps in which can be found tags indicating variation points from which different PUCs can be instantiated; the most common relationship is the relation between subsequent steps, such as those which form the main success scenario. Logical and **can be used to represent this kind of relationship, because every single step must be evaluated true to allow the entire PLUC to be evaluated** `true`**.**

A PUC is compliant to the family if, evaluating the tags expressions with the instantiation of variables given for that PLUC, all the tags are evaluated true. For doing that it sufficient the logical operator and**. Otherwise, the PUC cannot be accepted as belonging to the family: an inconsistent PUC has been identified.**

if (V0_tag && V1_tag && ... && Vn_tag)

then 'PUC is compliant'

else 'PUC is not compliant'

From the simple final logical expressions to be used to verify the compliance of a PUC to the family constraints those components having value false can be identified, and they are the single points determining the non compliance. Then it is simple to identify those instantiation to be modified to achieve the compliance to the family constraints.

It is easy to see that this expression evaluates to false: this means that a PUC with the variabilities solved with the above values does not describe any valid product of the family.

The structure of the tags allows those variability instantiations to be easily identified that determined the non-compliance of the derived PUC with respect to the PLUC. In this case the lack of compliance is due to the erroneous instantiation of V4.

7.4 Conclusions

In this chapter a methodology to express, in a formal way, requirements of products belonging to a product line is presented. It relies on a formalism allowing the representation of variabilities at the family level, and the instantiation of them in order to move to a single product. The instantiation of the tags (that can assume different values from a predefined range) in a PLUC-based requirements document determines the identification of the Use-Case-based requirements of a particular product (PUCs) belonging to the family with a configuration of the tags values.

Not all the possible instantiations of tags actually represent valid PUC-based requirements, because they do no satisfy given family constraints. The proposed method allows the formalization of these family constraints and the verification of the compliance to those constraints of a PUC-based requirements document.

One of the principal strenghts of the methodology described in this paper is the ease of inserting changes in family requirements expressed by means of PLUCs. In fact, if a tag is modified, because of the parametric nature of the approach, the effects of the modification affect only its definition and not its individual occurrences over the PLUCs. Moreover, if some new tags have to be added, the effort for doing that is mainly concentrated on the corresponding formal definition, and, once the new tag formula has been defined, the updating of the family requirements simply consists in the inclusion of the tag at the appropriate place of the affected PLUCs.

It is interesting to note how the described methodology can be used for supporting the impact analysis of possible new variabilities on the existing (or planned) products belonging to the family. When a new variable feature is to be added in the product line, it is of interest to evaluate its impact on the whole set of the family products. In particular, for evaluating if the new variability will determine incompatibility with some of the existing or planned products of the family a preliminary verification can be made adopting the verification procedure shown in section 3.2.

This approach is promising due to its simplicity and effectiveness for being implemented in an automatic way. In fact, it gives the advantage of an explicit identification of the variability points in a product line requirements document by means of the tags.

This characteristic may strongly facilitate the application of this approach in the industry because it allows the use of automatic tools for the identification of variabilities. In fact, suitable languages for expressing the different types of tags and products for make the verification automatic exist and they can be put together for building an environment where the proposed methodology can be implemented. That will be the object of the next steps our research activity will do. Indeed, the extention of the QuARS tool [5] with the aim to make it able to cover also the analysis of Use Cases and the automatic guided derivation of PUC belonging to the family is planned.

8. Conclusions

In this PhD. thesis techniques for the analysis of NL requirements are defined and their implementation is described. The work made in my PhD. course has been driven by a preliminary study of the state of the practice of the requirements process in the software industry.

In practice, tools and techniques for managing the requirements exist. They are mainly oriented to provide a framework where the requirements are defined, their configuration is managed and the distribution among the affected parties is controlled. There is a scarcity of automatic support for the quality analysis of NL requirements. Ambiguity analysis and consistency and completeness verification are activities usually made by single or multiple reviewers simply reading the requirements documents and looking for defects. This cleriacal activity is boring, and time consuming and often uneffective.

My research activity has been then oriented towards the definition and implementation of an original automatic tool able to perform an analysis of NL requirements in a systematic and automatic way. A quality model for NL requirements has been first defined and then an automatic tool, called QuARS (Quality Analyzer for Requirements Specifications), performing a quality evaluation against the defined quality model, has been implemented.

This tool allows the requirements engineers to perform an initial parsing of the requirements for automatically detecting potential linguistic defects that can determine ambiguity problems at the following development stages of the software product. This tool is able also to provide a support for consistency and completeness analysis by means of the View derivation. A View is composed of those sentences belonging to a requirements document and dealing with a particular argument, the availability of Views makes the detection of inconsistencies and incompletenesses easier because the reviewer has to consider smaller set of sentences where possible defects can be found with much less effort.

Research in the area of NL requirements analysis, experimentations with QuARS with real requirements documents taken from industrial projects and improvements to be made on the tool are the topics of my affiliation with the Software Engineering Institute of the Carnegie Mellon University – Pittsburgh, PA (U.S.A.) that has been established in early 2003 and is currently active.

The methods and techniques defined for the NL requirements analysis has been applied to a formalism that is going to be widely used in the industry: the Use Cases. Use Cases are a formalism, based on the NL language, that allows to capture functional requirements for software systems. They allow structuring requirements documents with user goals and provide a mean to specify the interactions between a certain software system and its environment.

The linguistic techniques defined to be applied to pure NL requirements are still valid for Use Cases, moreover, in this case, they allow a more precise analysis for consistency and completeness because the semantic data necessary for performing this kind of analysis can be derived in a more effective way.

The Use Cases formalism has been enhanced to make it suitable for representing requirements in the case of the Product Families (or Product Lines) development paradigm. This new Use Cases notation, called PLUC (Product Line Use Cases), allows to express the variability points in the Product Family requirements by means of tags expressed in a formal way. The adopted formalism for expressing the variability allows the verification of the compliance to the Family constraints of a single product.

In general, the initial approach to the NL requirements analysis offers many possibility to be applied, not only to the pure NL requirements, but also to other

formalisms (as the Use Cases). The research results achieved have been the object of several publications, but there are still many very stimulating research opportunities in this field, that I will address in the future.

9. References

1. **Abrams M., Jajodia S., Podell H., eds, Information Security – An integrated Collection of Essays, IEEE Computer Society Press, January 1995**

2. **Abrial JR. "The B Book - Assigning Programs to Meanings". Cambridge University Press, August 1996.**

3. **Alspaugh TA, Antòn AI. "Scenario Networks: A Case Study of the Enhanced Messaging System", 7th International Workshop on Requirements Engineering: Foundation for Software Quality REFSQ'01, Interlaken, Switzerland, June 2001.**

4. **Ambriola V, Gervasi V. "Processing Natural Language Requirements", 12th IEEE Conf. On Automated Software Engineering (ASE'97), IEEE Computer Society Press, Nov. 1997.**

5. **Ben Achour C, Tawbi M, Souveyet C. "Bridging the Gap between Users and Requirements Engineering: The Scenario-Based Approach" (CREWS Report Series 99-07), International Journal of Computer Systems Science and Engineering, Special Issue on Object-Oriented Information Systems, Vol. 14, N. 6, 1999.**

6. **Bertolino A., Fantechi A., Gnesi S., Lami G., Maccari A., "Use Case Description of Requirements for Product Lines", REPL'02, Essen, Germany, September 2002.**

7. **Bohem B.W. et alt., Characteristics of Software Quality, Elsevier North-Holland, 1978.**

8. **Bolognesi T, Brinksma E. "Introduction to the ISO Specification Language LOTOS". Computer Networks, 14 (1), 25-59, 1987.**

9. **Buglione L., Misurare il Software, Franco Angeli, 1999.**

10. **Byrne, E., "IEEE Standard 830: Recommended Practice for Software Requirements Specification," IEEE International Conference on Requirements Engineering, IEEE Computer Society Press, April 1994, p. 58.**

11. **Clements P.C., Northrop L. "Software Product Lines: Practices and Patterns". SEI Series in Software Engineering. Addison-Wesley, August 2001.**

12. **Cmap tool on-line: see http://www.yana.net/cmap/**

13. **Cockburn A. "Structuring Use Cases with goals", Journal of Object-Oriented Programming, Sep-Oct 1997 (part I) and Nov-Dec 1997 (part II).**

14. **Cockburn A. "Writing Effective Use Cases". Addison Wesley, 2001.**

15. **Conexor tool. See http://www.conexoroy.com/**

16. **Davis, A.M., Software Requirements: Objects, Functions and States. Prentice-Hall, 1993.**

17. **Dorfman, M., and R.H. Thayer, Software Engineering. IEEE Computer Society Press, 1997.**

18. **Dutoit AH, Peach B. "Developing Guidance and Tool Support for Rationale-based Use Case Specification", REFSQ'01, Interlaken, Switzerland, June 2001.**

19. **El-Emam K., Jung H-W, "An evaluation of the ISO/IEC 15504 assessment model", Journal of Systems and Software, n. 49, 2001, pp. 23-41.**

20. **Fabbrini F., Fantini E., Fusani M., Lami G. "Performing SPICE Assessments: Yet Another Tool". In Joint ESA - 3rd International SPICE Conference on Process Assessment and Improvement 17-21 March 2003 ESTEC, Noordwijk, The Netherlands.**

21. **Fabbrini F., Fusani M., Gervasi V., Gnesi S., Ruggieri S. "On linguistic quality of Natural Language Requirements". In 4 th International Workshop on Requirements Engineering: Foundations of Software Quality REFSQ'98, Pisa, June 1998.**

22. **Fabbrini F., Fusani M., Gnesi S., Lami G. "The Linguistic Approach to the Natural Language Requirements Quality: Benefits of the use of an Automatic Tool", 26th Annual**

IEEE Computer Society - NASA Goddard Space Flight Center Software Engineering Workshop, Greenbelt, MA, USA, November 27-29 2001.

23. **Fabbrini F., Fusani M., Gnesi S., Lami G. “An Automatic Quality Evaluation for Natural Language Requirements”, Seventh International Workshop on Requirements Engineering: Foundation for Software Quality, Interlaken, Switzerland, June 4-5 2001.**

24. **Fabbrini F., Fusani M., Gnesi S., Lami G. “Quality Evaluation of Software Requirement Specifications”, Proc of Software & Internet Quality Week 2000 Conference., San Francisco, CA May 31-June 2 2000, Session 8A2.**

25. **Fabbrini F., Fusani M., Gnesi S., Lami G. “Software Requirements Verification by Natural Language Analysis: a CNR Initiative for Italian SME’s” ERCIM News, Nr. 40, January 2000.**

26. **Fabbrini F., Fusani M., Lami G. "Assessing Software Process: the Rating Dilemma between Measurement and Human Judgment" Proc. of the 4rd International Symposium on Software Process Improvement, Recife, Brazil, 10-13 September 2002.**

27. **Fabbrini F., Fusani M., Lami G."Concepts and Practice of Software Certification" Proc. of the 5rd International Symposium on Software Process Improvement, Recife, Brazil, 3-5 Novembe r 2003.**

28. **Fabbrini F., Fusani M., Lami G., Sivera E. "A Methodological Approach to Improve the Software Acquisition Process in Automotive", The IASTED International Conference on Software Engineering", Innsbrick, Austria, February 16-19 2004.**

29. **Fabbrini F., Fusani M., Lami G., Sivera E. "Improving the Management of the Software Acquisition Process: a Methodological Approach in Automotive", 3rd Annual Conference on the Acquisition of Software-Intensive System", Arlington, VA (U.S.A.), January, 26-28 2004.**

30. **Fabbrini F., Fusani M., Lami G., Sivera E. "Managing Software Suppliers: An Experience of Process Assessment in Automotive" Proc. of the 3rd International Symposium on Software Process Improvement, Sao Paulo, Brazil, 17/20 September 2001.**

31. **Fabbrini F., Fusani M., Lami G., Sivera E. "Performing Process Assessment to Improve the Supplier Selection Process - An Experience in Automotive" European Software Process Improvement Conference EuroSPI2002, Nuremberg, Germany, 18-20 September 2002. p. 267-274.**

32. **Fabbrini F., Fusani M., Lami G., Sivera E. "Performing Software Process Assessment in the Automotive Environment". In 7th International Conference on Empirical Assessment in Software Engineering - EASE 2003. April 8-10 2003, Keele, UK.**

33. **Fabbrini F., Fusani M., Lami G., Sivera E. "Software Process Assessment as a Mean to Improve the Acquisition Process of an Automotive Manufacturer". In Software Process Improvment CMM&SPICE in Practice, Verlag UNI-DRUCK Ed. ,Munchen, Germany, 2002, pp. 142-154.**

34. **Fabbrini F., Fusani M., Lami G., Sivera E. "The Supplier Selection Process in Automotive: an Experience in Software Process Assessment" Proc. of the International Conference Software & System Engineering and their Applications - ICSSEA 2002, Paris, France, December 3-5 2002.**

35. **Fabbrini F., Fusani M., Lami G., Sivera E. "Using Software Process Assessment to Manage the Quality of Suppliers: an Experience in Automotive"Fifteenth International Software & Internet Quality Week Conference, San Francisco, CA 3-6 September 2002.**

36. **Fantechi A, Gnesi S, Ristori G, Carenini M, Vanocchi M, Moreschini P. “Assisting Requirement Formalization by Means of Natural Language Translation”, Formal Methods in System Design, vol 4, n.3, pp. 243-263, Kluwer Academic Publishers, 1994.**

37. **Fantechi A., Fusani M., Gnesi S., Ristori G. “Expressing properties of software requirements through syntactical rules”. Technical Report. IEI-CNR, 1997.**

38. **Fantechi A., Gnesi S., John i., Lami G., Dörr J. “Elicitation of Use Cases for Product Lines”, Fifth International Workshop on Product Family Engineering, PFE-5, Siena 4-6 November, 2003, to appear on LNCS Springer Verlag, 2004.**

39. **Fantechi A., Gnesi S., Lami G. “A Relation-based Approach to Use Case Analysis”. Proceedings of the 9th International Workshop on Requirements Engineering: Foundation for Software Quality, Velden, Austria, June 16-17 2003.**

40. **Fantechi A., Gnesi S., Lami G., Maccari A. “Application of Linguistic Techniques for Use Case Analysis”. Requirements Engineering Journal, Volume 8, Issue 3, pages 161-170, Spriger-Verlag, August 2003.**

41. **Fantechi A., Gnesi S., Lami G., Maccari A. “Linguistic Techniques for Use Cases Analysis”, Proceedings of the IEEE Joint International Requirements Engineering Conference - RE02. Essen, Germany, September 9 -13 2002.**

42. **Fenton N., Pfleeger S.L., Software Metrics: A Rigorous and Practical Approach, 2/e International Thompson Computer Press, 1997.**

43. Firesmith D. "Specifying Good Requirements", Journal of Object Technology, Vol. 2, No. 4, July-August 2003. ETH Zurich.

44. Fuchs N.E., Schwitter R. "Specifying Logic Programs in Controlled Natural Language", Workshop on Computational Logic for Natural Language Processing, Edinburgh, April 3-5, 1995.

45. Gennaro G., Lagelle D., Schabe H.. "Software Product Evaluation and Certification", Proc. Of Data Systems in Aerospace Conference, Nice (France), May 28 - June 1st 2001.

46. Goldin L, Berry DM. "Abstfinder, a prototype Abstraction Finder for Natural Language Text for Use in Requirements Elicitation: Design, Methodology, and Evaluation". First International Conference on Requirements Engineering, 1994.

47. Hailey V., "A comparison of ISO 9001 and the SPICE framework". In 'SPICE: The Theory and Practice of Software Process Improvement and Capability Determination", El-Emam K, Drouin JN, Melo W (eds). IEEE CS Press, 1998.

48. Halmans G., Pohl K. "Communicating the Variability of a Software-Product Family to Customers". Journal of Software and Systems Modeling, Springer, 2003

49. Harwell, R., et al, "What is a Requirement," Proc 3d Ann. Int'l Symp. Nat'l Council Systems Eng., (1993), pp.17-24.

50. Hofmann, H., Requirements Engineering: A Survey of Methods and Tools, Technical Report #TR- 93.05, Institute for Informatics, Zurich, Switzerland: University of Zurich, 1993.

51. Hooks I., "Writing Good Requirements", Proc. Of the Fourth International Symposium of the NCOSE, 1994, Vol. 2., pp. 197-203.

52. Horch, John W., Practical Guide to Software Quality Management, Artech-House Publishers, 1996.

53. IEEE Standard 610.12 Glossary of software engineering terminology, in Software Engineering Standards Collection, IEEE CS Press, Los Alamitos, Calif. 1990

54. IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications.

55. IEEE 1362-1998 IEEE Guide for Information Technology-System Definition -Concept of Operation Document

56. ISO/IEC 2382:1999 Information Technology - Vocabulary.

57. ISO 9001:2000 Quality Management Systems – Requirements TC 176 SC , 2000

58. ISO/IEC 9126-1: 2001, Software engineering - Product quality - Part 1: Quality model.

59. ISO/IEC TR 15504 (Parts 1-9), 1998

60. ISO/IEC TR 9126-2: Software engineering - Product quality - Part 2: External metrics.

61. ISO/IEC TR 9126-3: Software engineering - Product quality - Part 3: Internal metrics.

62. ISO/IEC TR 9126-4: Software engineering - Product quality - Part 4: Quality in Use.

63. ISO/IEC TR2 15504. "Information Technology – Software Process Assessment: Part 1–Part 9", ISO, Geneva, Switzerland, 1998.

64. ISO/TS 16949- Quality Management Systems - Automotive Suppliers - Particular Requirements for the Application of ISO 9001:2000 for Automotive Production and Relevant Service Part Organizations, 2002.

65. Jacobson I, Booch G, Rumbaugh J. "The Unified Modelling Language Reference Manual". Addison-Wesley, 1999.

66. Jazayeri M., Ran A., van der Linden F. "Software Architecture for Product Families: Principles and Practice", Publishers: Addison-Wesley, Reading, Mass. and London, 1998.

67. John I., Muthig D. "Tailoring Use Cases for Product Line Modeling", REPL'02, Essen, Germany, September 2002.

68. Jung H-W, Hunter R., "The relationship between ISO/IEC 15504 process capability levels, ISO 9001 certification and organization size: an empirical study", Journal of Systems and Software, n. 59, 2001, pp. 43–55.

69. Kamsties E, Peach B. "Taming Ambiguity in Natural Language Requirements", ICSSEA 2000, Paris, December 2000.

70. Kan, Stephen, H., Metrics and Models in Software QualityEngineering, Addison-Wesley Publishing Co., 1995.

71. Kassakian J. G., "Automotive Electrical Systems: The Power Electronics Market of the Future", Proc. Applied Power Electronics Conference (APEC2000), IEEE Press, 2000, pp. 3-9.

72. Kotonya, G., and I. Sommerville, Requirements Engineering: Processes and Techniques. John Wiley and Sons, 1998.

73. Krogstie J., Lindland O.I., Sindre G. "Towards a Deeper Understanding of Quality in Requirements Engineering". In 7th International CAiSE Conference, vol. 932 of Lecture Notes in Computer Science, pages 82-95, 1995.

74. Lami G. "Towards an Automatic Quality Evaluation of Natural Language Software Specifications". Technical Report. B4-25-11-99. IEI-CNR, 1999.

75. Leen G., Hefferman D., Dunne A., "Digital Networks in the Automotive Vehicle", IEE Computer and Control Eng. Journal, Dec. 1999, pp. 257-266.

76. Lehner F. "Quality Control in Software Documentation Based on Measurement of Text Comprehension and Text Comprehensibility". Information Processing & Management, vol; 29, No. 5, pp 551-568, 1993.

77. MacCall J.A., Richeards P.K., Walters G.F., Factors in Software Quality, Voll. I, II, III: Final Tech. Report, RADC-TR-77-369, Rome Air Deveopment Center, Air Force System Command, Griffis Air Force Base, NY 1977.

78. Macias B, Pulman SG. "Natural Language Processing for Requirement Specifications". In Redmill and Anderson, Safety Critical Systems, Chapman and Hall, 1993.

79. Mannion M., Camara J. "Theorem Proving for Product Line Model Verification", Fifth International Workshop on Product Family Engineering, PFE-5, Siena 4-6 November, 2003, to appear on LNCS Springer Verlag, 2004.

80. Meyer B. "On formalism in specifications". IEEE Software. January 1985, pages 6-26.

81. Mich L., Garigliano R., "Ambiguity measures in Requirements Engineering", Proc. International Conference on Software - Theory and Practice - ICS2000, 16th IFIP World Computer Congress, Beijing, China, 21-25 August 2000, Feng Y., Notkin D., Gaudel M., Publishing House of Electronics Industry, Beijing, 2000, pp. 39-48.

82. Minipar: http://www.cs.umanitoba.ca/~lindek/minipar.htm

83. Natt och Dag J, Regnell B, Carlshamre P, Andersson M, Karlsson J. "Evaluating Automated Support for Requirements Similarity Analysis in Market-Driven development". Seventh International Workshop on Requirements Engineering: Foundation for Software Quality, Interlaken, Switzerland, June 2001.

84. Nuseibeh B.A. and Easterbrook S.M. "Requirements Engineering: A Roadmap", In A. C. W. Finkelstein (ed) "The Future of Software Engineering". (Companion volume to the proceedings of the 22nd International Conference on Software Engineering, ICSE'00). IEEE Computer Society Press.

85. Paulk M., "Top-Level Standards Map: ISO 12207, ISO 15504 (Jan 1998 TR) Software CMM v1.1 and v2", Draft C (available at http://www.sei.cmu.edu/pub/cmm/Misc/standards-map.pdf), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1998.

86. Pfleeger, S. L., Software Engineering-Theory and Practice. Prentice-Hall, 1998.

87. Pohl, K., "The Three Dimensions of Requirements Engineering: A Framework and Its Applications," Information Systems 19, 3 (1994), pp. 243-258.

88. Pohl, K., "Process-centered Requirements Engineering", Research Studies Press, 1999.

89. Power N. "Variety and Quality in Requirements Documentation" Seventh International Workshop on Requirements Engineering: Foundation for Software Quality, Interlaken, Switzerland, June 4-5 2001.

90. Pressman, R.S. Software Engineering: A Practitioner's Approach (4 edition). McGraw-Hill, 1997.

91. Quality System Requirements (QS-9000 Third Edition) Version 03.00, DaimlerChrysler, Ford Motor Company and General Motors Quality Publications, March 1998.

92. Regnell B., Beremark P., Eklundh O. "A Market-Driven Requirements Engineering Process: Results From an Industrial Process Improvement Programme", Requirements Engineering, 3(2), 1998, pp. 121-129.

93. Richards D., Boettger K., Aguilera O. "A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices", LNAI 2557.

94. Rolland C., Proix C., "A Natural Language Approach for Requirements Engineering". AISE'92, LNCS 593, Springer-Verlag, 1992.

95. Rosenberg L., Hammer T.F., Huffman L.L. „Requirements Testing and Metrics", 15th Annual Pacific Northwest Software Quality Conference, Utah, October 1998.

96. Siddiqi, J., and M.C.Shekaran, "Requirements Engineering: The Emerging Wisdom," IEEE Software, pp.15-19, 1996.

97. Sommerville, I. Software Engineering (5th edition), Addison-Wesley, 1996.

98. Sommerville, I., and P. Sawyer, Requirements engineering: A Good Practice Guide. John Wiley and Sons, 1997

99. Sommerville, I., and P.Sawyer, "Viewpoints: Principles, Problems, and a Practical Approach to Requirements Engineering," Annals of Software Engineering, 3, N. Mead, ed., 1997.

100. Spivey JM. "The Z Notation: A Reference Manual", 2nd edn., London Prentice-Hall, 1992.

101. Suhl C. "RT-Z: An Integration o Z and timed CSP". In *Integrated Formal Methods (IFM'99)*. Springer-Verlag, 1999.

102. SyTwo on-line. See: http://www.yana.net/sytwo/index.html

103. Thayer, R.H., and M.Dorfman, Software Requirements Engineering (2nd Ed). IEEE Computer Society Press, 1997.

104. The Motor Industry Software Reliability Association "Devebpment Guidelines For Vehicle based Software", , 1994. Published by MIRA. ISBN 0952415607

105. Van der Linden F. "Software Product Families in Europe: The ESAPS & Café Projects" IEEE Software, Vol 19, n. 4, July/August 2002.

106. Welch B. "Practical Programming in Tcl and Tk" second edition Prentice Hall 1997.

107. Wilson W.M., Rosenberg L.H., Hyatt L.E. "Automated quality analysis of Natural Language Requirement specifications". PNSQC Conference, October 1996.

108. Wilson W.M., Rosenberg L.H. Hyatt L.E. "Automated Analysis of Requirement Specifications". Proceedings of the Nineteenth International Conference on Software Engineering (ICSE-97), Boston, MA, May 1997.

109. Wing J.M., Woodcock J., Davies J. (eds.) FM'99 – Formal Methods, vol. I and II LNCS 1708, 1709, Springer.

110. Zowghi D, Gervasi V, McRae A. "Using Default Reasoning to Discover Inconsistencies in Natural Language Requirements", Proc. of the 8th Asia-Pacific Software Engineering Conference, December 2001.

Page: 80

[AM1]How about: “a use case describes the interaction (triggered by an external actor in order to achieve a goal) between a system and its environment”?