Software Requirements Engineering: An Overview

Daniel Jitnah
Jun Han
Phillip Steele

Peninsula School of Computing
and Information Technology
Monash University

Abstract

Software requirements engineering can be considered as the process by which the features of software systems as perceived by the user are established. Software systems are destined to be embedded into organisational settings, which impose constraints on performance and operational characteristics. Determining how the software should behave within those constraints requires a thorough understanding of the software functionalities, its attributes, the complexity of the interaction between them, and its interaction with the user environment. The primary source of information is the user (organisation). However it is often the case that users inadequately elicit software requirements and in a form inappropriate to the software developers. Software requirements have to be established by an iterative refinement process of elicitation, analysis, modelling and representation.

An important but intractable issue is that of knowledge representation and management, and specially domain knowledge. Domain knowledge forms the basis for establishing a common universe of discourse (UoD) between users and analysts, for mutual understanding, reference and validation of requirements. There is an urgent need for effective support mechanism for capturing and managing such domain knowledge. Any support method must allow the interrelation between the domain knowledge and the software requirement information to be represented, so that inadequacies of the requirements can be recognised and dealt with accordingly.

1 Introduction

Software systems analysts have to identify the purpose for which software systems are to be developed. In general, the purpose is to perform certain tasks as determined by the software users. Hence software systems have to provide functionalities that allow them to perform the tasks for which they have been designed. Furthermore software systems are destined to be embedded into organisational settings, consisting of human beings, machines and possibly other software systems. The organisations impose constraints on the operational and performance characteristics of the software. Human beings benefit from the applications of software systems in data processing, information management, machine control systems, communication networks etc. The useability of software systems depends on the perception of their users as to what the software should do, can do and how well they fit into the users' environment. Users therefore require that software behave in ways that meet their perceptions. Software requirements engineering (SRE) is the process through which those user requirements are identified and then expressed in a document: the software requirements specification document (SRSD). (Note that in the current literature, the words "requirements specification" are sometimes used to mean either the "process" or the "document" produced at the end of the process. In this document the terms SRE and SRSD will be used as defined above to avoid any confusion).

SRE must therefore allow the requirements to be identified as they are perceived by the users, most likely to be human beings. SRE can be regarded as the process of mapping the framework of ideas in the non-technical, informal (in the computing sense) universe of discourse (UoD) of the user onto a technical, semi-formal and formal software UoD. During the process, the perceived requirements will be identified and analysed. Unfortunately this has proved to be a complex and intractable task. Surveys and statistics abound to show the shortcomings of the recent and current state of practice of SRE (Davis 1990).

Although the problems have been recognised for a number of years (IEEE 77), it is only recently that the scientific research community has started to focus their efforts on software requirements engineering, as a specific software engineering research area, in its own right. Indeed the scope of the tasks comprising SRE, in relation to the entire software engineering process, is as yet to be clearly defined. It is nevertheless agreed upon that RE is mostly concerned with the early concept identification and analysis phases, detached from the technical characteristics and detail design issues of software development(Davis 1990).

In this paper relevant matters in SRE are reviewed: they are wide ranging - and need be so, since SRE bridges the gap between the unconstrained software user world and the more constrained software developer world. Requirements engineering is first situated in relation to Information Systems and Software Systems. Issues relating to activities of software requirements engineering: requirement elicitation and requirement analysis are then discussed. Commonly used methods for requirement elicitation are described. Particular emphasis is placed on the importance of domain knowledge, both in relation to requirement elicitation and analysis. We also discuss support methods in terms of formalisation of requirement engineering process, and in terms of management methods and tools for software requirement information.

2 Preliminaries

2.1 Requirements for software systems and information systems

While software requirements engineering in general refers to the early concept identification and analysis stages, it is also necessary to distinguish between requirements in relation to information systems and in relation to software systems. The following example illustrates this distinction.

Consider a manufacturing organisation X. It is generally accepted that X will have as main objective the maximisation of profit on its operations. In order to achieve its goal, X adheres to organisational structures and procedures. Some procedures may aim at X being able to minimise the stock of raw material at any time, and that the right quantity of raw material be ordered in time. This requires that X has an up to date record of its stock, has accurate record and monitoring procedures of the usage pattern of the raw material, has accurate trace of purchase orders, etc. These are information systems requirements.

If it is decided that the raw material inventory, usage monitoring and purchase management are to be automated, then those requirements may then be considered as high level software requirements. X software systems requirements will include an inventory management system. These will in turn consist of methods to record the movement of raw material into the factory warehouse, onto the production line, return to suppliers (if defective), methods to query the inventory database, methods to record and follow-up purchase orders, etc. The software requirements may be expressed in various and complementary forms, such as plain English, semi formal forms including tabular representation, flow diagrams, process charts, entity relationship diagrams, etc. The software requirements may also include specifications regarding the characteristics of the computer systems it is to be used on.

Software systems requirements engineering necessarily relates to the needs of the organisation in terms of achieving certain results by the use of a software system. The development (or possibly modification) of a particular software system usually follows the SRE exercise. Software requirements are requirements that enable the software systems to

perform its function. Information Systems Requirements do not necessarily relate to any software system. They are requirements that relate to the information needed to enable the organisation to function. They are requirements of the organisation. When the software is viewed as being part of the organisation, then there will be an overlap in the software requirements and the information systems requirements.

2.2 Activities of software requirements engineering

Software requirements engineering can be regarded as consisting of a set of interrelating activities that result in the production of a SRSD. SRE consists of requirement elicitation activities, requirement analysis activities and requirement specification activities. This decomposition of the SRE process has to be taken as a convenient description of necessary tasks that are carried out These tasks are not necessarily performed in any order or at distinct periods in time. However a particular item in the SRSD will have been elicited, analysed and specified. In fact, a single activity may be seen as requirement elicitation and requirement analysis at the same time. Requirements are often synthesised during an iterative elicitation and analysis cycle. There is no necessary clear distinction between the elicitation and analysis phases. For example a discussion between the analyst and the customer can reveal particular characteristics of the requirements (analysis) and stimulate the customer to enunciate new or modified requirements (elicitation). This point is discussed again later.

It is appropriate to explain what is considered as being *an item of requirement* as used throughout this document. *An item of requirement* refers to a *capability* provided by the software system and which can produce a particular result (eg: an input, an output, or a data transformation), that is *observable* by the user. The characteristic that a requirement relates to an *observable* result is necessary, as it provides a means of verifying that the software behaves as expected by the users. Each capability can be considered on its own, and can be implemented independently from other capabilities provided by the system. This description is similar to that provided by Ramesh and Dhar (1992), and also used in the IEEE Standard 729 (IEEE, 1993). Note that although capabilities may be considered independently, they may share common sub-processes(eg, software subroutines) as specified during the software design and implementation phase. However the design of the software should not affect the actual result produced by the capability, but should only be considered as a matter of convenience and efficiency, given the particular software development platform chosen. In fact, it is generally accepted that design consideration should not be considered during the SRE process. The set of all such requirement items form the requirement specification for the software system.

3 Requirement Elicitation

Requirement elicitation is the process of identifying items of information that determine the desired characteristics of the software system. Those characteristics determine the behaviour of the system in response to user input. More importantly, the software system must have effects on its environment that are satisfactory to the user. Its behaviour must meet the users' expectations. The users' perception of the system's behaviour depends on the users' experience and background knowledge, and the environment within which the system is embedded. It is therefore necessary to consider information about the application domain and about the users of the software system as well. During requirement elicitation such knowledge must be obtained. Domain knowledge is necessary to enable the analyst to reach a common ground for understanding the user world.

3.1 Elicitation methods

Goguen and Linde (1993) discuss various methods of acquiring requirement information. These methods are based on commonly accepted social activities and behaviours, and are briefly described below.

Introspection. This is the process by which the software developer *imagines* the system that would perform the required task. This approach can be effective when the software developer, also described as "super-designers", is the actual end user or is an application domain expert (Lubars, Potts and Richter, 1993). However introspection has obvious limitations, when the software developer has no or limited knowledge of the area of application. Introspection is however still used in the specification of software features that the user may have little experience with, such as the user interface. It is important that the software developer does not make undue assumptions regarding the user perception of such features, and attempts to validate those that are made. Use of software prototypes can be used for such validation activities.

Interviews. Questionare interviews and open-ended interviews can be used to identify requirements information. Questionare interviews consist of a pre-determined set of questions that the analyst poses to the users and their responses are recorded. Questionnaire interviews limit the topics of investigation to those raised by the questions. They do not facilitate the discussion of other unplanned issues. Open-ended interviews allow the participants to focus on any matter that may be raised. Interviews are useful in allowing domain knowledge to be identified. However, interviews depend on the ability of the interviewees to express information adequately. Interviewees elicit requirements as they perceive them, and may fail to explain their needs from a detached and global perspective. Furthermore, it is observed that interviewees are often unable to elicit tacit knowledge satisfactorily. Interviews are also inadequate, because they are constrained conversation methods (consisting of 'turn-taking', question and answer protocol), which restrain the expressive abilities of the participants.

Focus groups. Focus groups are similar to interviews, but allow for spontaneous conversation. The analyst sets a topic of interest and lets a group of concerned users discuss it openly. Focus groups have proved popular in information systems application (JAD, RAD). An important factor in the success of focus groups is the choice of participants. The analyst must take into consideration the positions of the participants in the organisation regarding their authorities and abilities to discuss matters of concern.

Protocol analysis. This is the process where information is gathered about tasks, by observing users engaged in performing the tasks, and asking them to "talk aloud" their thought processes. Goguen and Linde indicates that Protocol Analysis has not been very successful. Users can experience difficulty describing what they are doing satisfactorily. The use of oral communication for this purpose is considered as being unnatural.

Other methods used for requirements elicitation are described as follows:

Prototyping. In the context of requirements elicitation, prototyping is used to provide to the user first hand experience of how the software system will behave. A software prototype is produced and presented to the user for evaluation. Inadequacies in the systems functionalities can be identified, and corresponding improvements are carried out until the user is satisfied. Prototyping is particularly useful when establishing requirements for user interfaces.

During the development of a software prototype, design decisions are often made early in the software development process, and considerable effort from the software developer may be required. However, it is generally argued (Davis, 1990) that design considerations are inappropriate during the SRE phase. A "throw-away" prototype (designed to be discarded after use), can be used, with care, to validate some functionalities of the software system. But users may draw incorrect conclusions about other characteristics of the software, based on their evaluation of the prototype presented to them. Hence the exact purpose of the use of the prototype needs to be clearly established and understood by both the software developer and the users.

Animation. In an attempt to resolve some of the problems associated with software prototype, and specially to reduce their development effort, Kramer and Ng (1988) propose

the use of a*nimation* techniques for software requirements elicitation. Software animation has similar aims as prototyping. However during animation, features of the software that are difficult to specify are "role-played" by human beings Furthermore, animation emphasises the effects of and the relationships between processes, instead of how the results are achieved. For this reason animation is more appropriate for early requirements elicitation purposes. Animation also allows for more flexibility, as processes can be easily modified and reorganised, compared to the process of modifying source code of software prototypes .

Scenario analysis. Potts, Takahashi and Anton (1994) discuss the use of scenarios for requirement elicitation. Used in conjunction with prototyping and animation, scenario analysis allows the software developer to investigate system responses to various conditions. The analyst effectively poses "IF condition THEN response" scenarios to the user. In this way the user is required to elicit the behaviour of the system in specific situations. By appropriate choice of scenarios, the functional characteristics of the software can be comprehensively identified.

Ethnographic study. Ethnography is commonly used in the study of anthropological and social phenomena. It involves a researcher spending time living within the society and observing its practices. In the context of SRE, a similar ethnographic study involves the software developer (team) observing and participating with the users in the regular daily activities (Sommerville, Rodden, Sawyer, Bentley and Twidale, 1993). The software developer is able to get first hand understanding of the organisation setting in which the software is to be used, and how the latter relates to the work schedule and aims of the organisation. This technique also enables the software developer to gain substantial background knowledge about the organisation, which would otherwise be hard to acquire. The success of this approach to requirement acquisition depends on the willingness of the user organisation and its employees to cooperate with and to accept that the analyst gets involved in their work place. In ideal situation, the analysts and the users are able to reach a common ground for understanding concepts and issues relating to the requirements. From the analysts point of view though, the risk exists that they become so much involved inside the organisation, that they lose objectivity in their observation and ability to analyse the user requirements. For example, undue emphasis can be placed on one aspect of the software, because it raises sensitive issues among the employees, but which has little bearing on the effectiveness of the software as a whole.

3.2 General comments

In the requirements elicitation methods discussed above, the perception, interpretation and expression of relevant information by users, and subsequent interpretation and recording by the analyst, are crucial issues. Misinterpretation of the requirements can be caused by the analyst failing to grasp and record the context within which users express their perception of the software systems behaviour. Recording the context is necessary. One analyst may correctly understand the requirements at the time they are identified, as she/he is in the thick of the discussion. However, at a later stage of the requirements elicitation or software development, those requirements may be interpreted differently. This is a problem associated with recording and reporting of events in general, and is commonly encountered in reporting of news events for example. It is often the case that news reporters will later only report their version of the facts, which can be a distorted view of the event as it occurred.

4 Requirement Analysis

4.1 Inquiry cycle for requirement analysis

Earlier in this document, it was mentioned that the discussion of SRE in terms of the three distinct activities: requirement elicitation, requirement analysis and requirement specification, was a convenient modelling approach. In real life situations, the three activities are more

likely to occur concurrently throughout the SRE process. SRE is viewed as an interactive process involving interested participants: software developer, users and customers. This interactive SRE process description closely follows that of normal human conversation as they occur in everyday life. Furthermore, requirements are considered to be gradually synthesised and refined. This is to be distinguished from the pre-supposition that requirements are there, ready to be extracted from the users - and in fact it is common to come across users who are uncertain about what they really want from a software system. This is why Goguen (1994) describes software requirements as being "*emergent".*

Potts, Takahashi and Anton (1994) introduce the *Inquiry Cycle for Requirement Analysis*. They view the requirements engineering process as consisting of a cycle during which items of requirement are elicited and then subject to discussion immediately. Discussions consist of statements of positions, question and answer dialogues, and justifications of decisions. During this process, modifications to the initial requirements are requested, hence stimulating an evolutionary approach to SRE. Potts et al. also discuss hypertext based support for recording the inquiry cycle. In addition they discuss the use of scenario techniques to prompt discussions about the requirements. Potts et al. argue that the inquiry cycle allows for flexibility in the SRE process. This is achieved by accommodating and recognising the importance of spontaneous inputs from participants, about issues that may be overlooked otherwise.

4.2 Requirement analysis vs requirement elicitation

Requirements analysis can be considered as the process by which the adequacy and characteristics of the identified requirements are established. The requirements are examined to determine whether they coherently describe the behaviour of the software system. The aim of requirements analysis is also to reveal inadequacies (if any) in the elicited requirements that can prevent the software from meeting user satisfaction and cause difficulties later in the software development process. During requirement analysis it is necessary to identify relationships and contradictions between the items of requirements.

In the same way as the requirement elicitation process needs to take into account the context and background of the users, the analysis of software requirements needs to be performed within the appropriate framework of concepts that matches that of the users. If the software developer does not possess or understand those concepts, it will be necessary to obtain them. Such concept acquisition will be achieved (at least partly) during requirement elicitation, as described earlier. This strongly suggests that requirement analysis and elicitation can be considered as concurrent and closely related activities, dealing with the same or closely related items of information. The distinction between the processes of elicitation (Collins English Dictionary: elicit = to bring to light, to give rise to) and analysis (Collins English Dictionary: analysis = division of ... abstract whole into its constituents to determine their relationship or value, separation of a concept from another that contains it) may reside in their purposes: elicitation is to bring to light the requirements, and analysis is the bring to light the characteristics of those requirements. But we can consider that those identified requirement characteristics are also items of requirement, ie, a refined view of requirements. In this sense elicitation and analysis can be viewed as interleaved processes.

Still, it may be possible to distinguish between the two activities in terms of the methods and approaches used. Elicitation is an initial mostly informal concept "bringing to light" process, compared to analysis which uses more formal methods. But yet again this distinction is debatable as it is common to regard analysis as also being a more formal concept "bringing to light" process, starting right at the start of the SRE process, ie: with no distinct elicitation activity taking place. So the distinction between analysis and elicitation resides in the conceptual framework within which those activities are described, or in the issues that matter. Throughout this paper we will adopt this conceptual distinction between analysis and elicitation. But we also pose the question whether such a distinction is useful. We do not attempt to answer this question as yet.

Below is a brief discussion of issues of concern during requirement analysis, relating to the characteristics of the requirement, as interpreted in the user world .

4.3 Completeness

Software requirements need to be complete. They should describe all the functions that the user expects the software to provide. Otherwise, it is possible that the software developer may make assumptions regarding the requirements, which may be invalid to the user. For example, a requirement may state that a set of input values must fall between a specified range to be acceptable, but it does not specify what has to be done if an input value falls outside the range. The software developer may assume that the value is to be ignored, while the user expects that an error condition is reported. Completeness of requirements specification is often established as a gradual process during the later stages of the software development.

Furthermore, it may be necessary to identify what a software must not allow to occur, as well as what it should do. This issue is discussed by (Davis, 93), using the following example, referring to a telephone system, where a requirement may state:

> If a party A calls party B, then party B's phone will ring.

Davis argues that this requirement can be met by allowing all phones in the system to ring whenever A calls B! This problem can be resolved by stating that only B's phone should ring and no other ones. But this too will be unsatisfactory, as there may be genuine reasons why other phones than B's should also ring if A calls B (eg, call diversion).

This example shows that it may be impossible to completely specify software system requirements. There is an infinite number of things that a software must not do and there are equally many functionalities that a software may allow to occur, without affecting its purpose. There is hence a crucial need to establish a common Universe of Discourse (UoD) between the user and software developer, which forms the basis for mutual understanding and resolves the necessity of completely specifying software requirements.

4.4 Ambiguities

Ambiguities in requirements can occur as a result of ambiguities in the way they are expressed in natural language: *ambiguity of expression*. Natural language statements may often be subject to different interpretations. For example a requirement specification may state that: "If the pressure exceeds 100 KPa or the temperature exceeds 200 C and the pressure release valve monitor detects a malfunction, then raise the alarm". This requirement is ambiguous because it can be interpreted as saying that if the pressure release valve is malfunctioning, and the temperature is 199 C, then the alarm will not be raised.

Ambiguities can also arise of inadequacy of specification about how a particular event should be handled: *ambiguity of requirement content*. For example an air traffic control (ATC) software may be required to monitor up to 100 aircraft simultaneously, but it must also not allow a 101st aircraft to fly into the controlled airspace un-monitored. It is easy in similar circumstances for the client/user to assume the case of the 101st aircraft to be self evident, and to fail to elicit that particular requirement. The case of the 101st aircraft can be considered as ambiguous since it has not been specified what step is to be taken in the event of an extra aircraft entering the space.

Ambiguities in requirements specification can usually be resolved by considering additional information, which invalidates the other possible ways of interpreting the original statement. However, ambiguities may be difficult to detect, because they may not prevent the production of a working software, as the software developer may not realise that they have misinterpreted the requirement until after the software is produced and tested.

4.5 Inconsistencies

Inconsistencies in software requirements can appear as requirements relating to situations that are mutually exclusive. One particular statement of requirement may itself be inconsistent, but a more likely situation is when two separate items of specification are mutually conflicting, while each one of them considered separately is valid. For example, in a system controlling the temperature of a furnace, the statements:

i. If the temperature of the furnace exceeds 200 C, reduce the power to the heating element.

ii. If the temperature of the furnace falls below 210 C, increase the power to the heating element.

can be regarded as being inconsistent requirements. To understand why these statements are inconsistent, it is necessary to understand the context within which they have been expressed. In the present context, it is assumed that the 2 statements are elicited by 2 separate persons, at 2 different points of time. Statement (i) relates to a situation where the furnace must temperature must not be allowed to rise above 200 C, and where a lower temperature is in fact desirable. Statement (ii) relates to a situation where the furnace temperature must not be allowed to fall below 210 C, and where a higher temperature is desirable. Another important property of the above system is that the effect of reducing or increasing the power is directly related to the trigger condition for the action, ie, reducing the power will reduce the temperature and will therefore change the trigger condition. Assuming that the furnace temperature starts a normal room temperature: say 25 C, and if statement (i) is adhered to, it can be seen that a temperature of 210 C will be never be reached, hence statement (ii) will never be satisfied.

Such inconsistencies may creep into the actual software, as the two functionalities may be coded by different programmers, and remain undetected until well into the testing stage or real life application stage. Notice that in the above case, the functionality required can be successfully coded into a software system. (It is only required for the software to initiate responses to tests on range of values.) There is no requirement for the software to model the fact that the quantity being measured and tested (temperature) is directly related to the response initiated (reduction or increase of heating element power, ie, changing the power will change the temperature) and that the temperature starts at room temperature.

In fact the following example illustrates a similar functionality (ie: initiate responses to tests on range of values) but without the situation where the tested quantity (value of share) and response (decision to sell or buy) being directly related, and with initial value of tested quantity being unknown. The example may not be considered as being inconsistent.

i. If the value of a share exceeds $20, then sell shares.

ii. If the value of a share falls below $20, then buy shares.

The point to make is that the information that distinguishes the 2 cases discussed above and that determines whether the requirements are inconsistent are not items of information that are usually modelled and represented by the software system. They are application *domain information* and *elicitation context information*, that may need to be explained, and necessary to establish the validity of the requirements. This indeed shows the intricate relationship between software requirements and the domain of interpretation within which they are elicited.

4.6 Correctness

Establishing the correctness of the software requirements is the process of determining if the requirements will enable the organisation's objectives to be satisfied. (This needs to be

distinguished from establishing whether the requirements are correctly implemented during the software development process.) A requirement may be successfully satisfied by a software system, but the requirement itself may be incorrect and unsatisfactory to the organisation. For example a software system may be specified as being required to monitor and hold the temperature inside a furnace fixed at a predetermined value to within 5% deviation. This may be successfully achieved by the software system. However this requirement in itself may be unsatisfactory, as in this case, in order to meet specific manufacturing quality standards, the furnace temperature must be kept accurate to within 1% deviation. It is reasonable to expect the analyst to accept the first requirement information as being correct, in the absence of contradictory information from a different source. Incorrectly elicited requirement may be identified when conflicting information are obtained. Hence when the production manager may later indicate that the acceptable deviation to the furnace temperature must be kept within 1% because of more stringent quality standards, the analyst's task therefore is to query this apparent conflict and resolve which is the correct requirement.

Another example of a subtle incorrect requirement is in the case of the air traffic control software: "...*required to monitor up to 100 aircraft simultaneously",* since what is more correctly required is for the system to be able to monitor 100 aircraft according to normal procedures, and consider all extra aircraft as exceptions and handle them differently. As the case of the 101st aircraft shows, the ATC must in fact be able to monitor more than 100 aircraft, at least to be able to detect the extra aircraft!

The above examples, once more, show the need for a common ground for understanding to be reached between the user and the software developer. The latter must understand the implication of each item of requirement to the operation of the organisation. Items of requirement need to be interrelated and compared so that potential conflicts can be detected.

4.7 Relationships between requirements

Items of software requirements can be related in several other ways. Relationships between requirements can arise because of commonalities in the items of discourse. In the following examples, the common discourse item is the accuracy of computation expressed in decimal points.

A requirement R1 necessitates that requirement R2 be met.

For example, a requirement that the result of a mathematical computation be accurate to 3 decimal points (R1), necessitates that the input data be at least accurate to 3 decimal points (R2).

A requirement R1 includes (sub-)requirements R1a, R1b, ...

For the same example, R1 includes the requirements that each individual mathematical operation also produces sufficiently accurate results.

A requirement R1 meets R3

R1 meets "R3: results of computation be at least accurate to 2 decimal points".

A requirement R1 conflicts with R4

A requirement R4 that input data be accurate to 1 decimal point conflicts with R1.

R1 has side effect R5

R1 has side effect "R5: format of report printouts be able to show values accurate to 3 decimal points".

A requirement R1 is similar to R6

For a requirement R6 different from R1, stating that a different computation be also accurate to 3 decimal points, R1 is similar to R6. Similarity between items of requirement can lead to reuse of software artefacts (design, coding, etc). When identifying similarity between requirement items, it is necessary to establish the criteria on which the similarity is based. In this example, the requirements are similar in the accuracy of computation.

Commonalities in requirement items allow them to be associated and further investigated for instances of conflicts, similarities, etc. Often commonalities can only be recognised by experienced and aware users and analysts. They can appear as synonymous words used in different situations and can span over several (sub)-systems. Commonalities also appear as items of information that are closely related to each other, eg, accuracy and decimal points, as in the above examples.

Once instances of commonality have been recognised, it is necessary to record and represent them in such a way that they are easily recognisable during later sessions of requirement analysis. Recording relationships between items of requirement is made difficult by the fact that they can be elicited by different persons at different times. Furthermore relationships between requirements can be difficult to establish, specially when they relate to matters that have varying importance and relevance with regard to the topic of analysis. Hence the commonality between the discourse items relating to the requirements may not be emphasised.

Mylopoulos, Chang and Nixon (1992) discuss other types of relationship between requirements, although their discussion applies to information systems requirements in general. Requirement information is related by typed links. For example, a particular item of requirement may be related to a sub-requirement by a "satisfycing" link, indicating that meeting that sub-requirement is a necessary step in meeting the parent requirement. Hence a parent requirement is met when all its sub-requirements can be linked to it by "satisfycing" links. They also indicate how a failure to establish a "satisfycing" link (shown as a "denying" link) can be used to represent the inability to meet a particular item of requirement. Mylopoulos et al. also discuss the idea of correlation rules to indicate the nature of relationships between requirements. Such linking mechanism can prove useful in the analysis and validation of software requirements.

4.8 Requirement traceability

There are several issues of concern during the SRE process, as indicated earlier, and which can impact on the validity of the SRSD. As we have seen, capturing application domain knowledge is necessary in enabling a common framework of concepts to be established, to allow agreement to be reached between parties concerned. The SRE process occurs over a period of time, and at each instant only a small part of the software requirements will be in focus. In this process the interrelationship between the items of requirement and related information may be overlooked. The importance of maintaining requirement traceability has been discussed in length by Gotel and Finkelstein(1994) (not only in relation to SRE but in relation to the whole software engineering process). Software requirements traceability arises in terms of pre-requirement (requirement production) and post-requirement (requirement deployment) traceability. Both aspects need support mechanism. Support tools in the form of database systems, hypertext and hypermedia systems have been proposed (Potts, Takahashi and Anton, 1994).

There are a number of requirement traceability types that need to be maintained:

Inter requirement traceability
Requirement to sub-requirement (and vice-versa)
Requirement to source of requirement (likely to be a user)
Versions of requirement specification statements
Requirement to requirement rationale
Requirement to annotation information

The last two mentioned types of requirement traceability are probably the most difficult to support, in view of the potential informality of the information that is involved. Yet they are the most important ones in terms of capturing the "unstated" information**.**

4.9 Functional vs non-functional requirements

It is common practice to distinguish between functional and non-functional characteristics of software systems. Functional requirements refer to tasks that a software is expected to perform. They describe actions and processes resulting in data transformation. Functional requirements are met by capabilities provided by the software system.

Non-functional requirements describe other characteristics and relate to constraints that has to be met by the software. Non-functional requirements are met by allowing the software to satisfy conditions constraining the execution of the functional requirements and other conditions. Constraints are often external to the system and outside the software developer's control.

However, while non-functional requirements do not describe processes, their violations can negate the ability of a software to meet the specified functional requirements. For example a software system may have as functional requirement the ability to produce a summary listing of a shop sales level for a day. A corresponding non-functional requirement is that this report must be produced within 1 hour of shop closure. If it is not possible to meet this time deadline (eg, because sales data take longer to acquire), although the processing capability exists to produce such a listing, the stated functional requirement will not be satisfied.

4.10 Functions vs structures

The concepts of functional and non-functional requirements provide useful abstractions for describing software requirements. However they fail to bring about the necessary interaction between the software and its environment. Soares (1994) discussed software requirements in terms of functions and structures. This is particularly useful when dealing with systems that consist of material flows, energy flows and process/manufacturing control systems in general. In such contexts, the system components and the interaction between them define the *structure*. For example, a computer controlled drilling system can consist of a motor, the gearing mechanism, the drill-tip, the control switches and the control software. Together they form the structure of the drilling system. This structure is capable of carrying out certain actions, ie, the structure has a functioning, determined by the interaction between the components. These actions can be used to achieve specific purposes. The purpose is external to the system, and is attributed to it by the user, by the way it is put to use. The functioning of the drilling system (determined by its structure) however is the same whatever purpose it is used for. The function of the drilling system is its purpose considered within the (larger) structure consisting of the drilling system (itself considered as a sub-structure) and its environment. It is only possible to specify the *function* of a system within a (larger) structure, and changing such structure will change the function of the system (but may not change the functioning of the system, which is considered within the (smaller) structure of the system components). If the function of the drilling system is to drill holes in metal blocks, then such function can only be achieved within the structure consisting of drill and metal blocks. If however the (unlikely) function of the drill is to create deafening noise, then that function can only be achieved within the structure consisting of drill, air and listener! The same argument

applies to the control software, ie, the function of a software system can only be interpreted in relation to the structure within which it is functioning.

Consequently, this discussion suggests that SRE needs to consider the software requirements within the appropriate context of analysis and appropriate structure. It is not sufficient (although necessary) for the functioning of a software to be eventually established from the SRSD, but it must also ensure that the software (by its functioning) achieves its function, ie: its purpose. Software requirements must therefore be viewed from a wider perspective, taking adequate consideration of the application domain within which the resulting software system is to be used.

5 Representing and Specifying Software Requirements

Underlying the software requirements engineering process are representational and specification methods. Elicited information needs to be recorded and represented in a form that will successfully capture the important facts and the relationships between them, and facilitate analysis of the identified information. The software requirements must also be specified in a document (the SRSD) which can be used for various purposes, such as source document for code design and development, and possibly as a contractual document. Ideally a single representation method should meet all three aspects of requirements engineering: elicitation, analysis and specification. However no representation methods, of the author's knowledge, meet this characteristic. Software requirements are commonly expressed in semi-formal or formal languages.

5.1 Representation in a modelling language

The purpose of representing software requirements information in a modelling language is to enable requirement analysis. Various methods for requirements modelling have been proposed. Many of them are based on general software systems and information systems modelling methods. They can be categorised as entity and object oriented methods, function oriented methods and state oriented methods. These include the use of representation techniques like entity relationship diagrams, data flow diagrams, object oriented analysis, structured analysis and design technique (SADT). The applicable methods are reviewed in Davis (1990). The analysis technique allows the analyst to obtain a thorough understanding of the environment (objects, processes and states) within which the software is going to be deployed.

Furthermore, such software requirement modelling methods provide frameworks of concepts, that are used to model the characteristics of the software system. For example the object oriented (OO) method, provides abstraction for modelling objects in the real world in terms of object classes, hierarchies of object classes, and associated characteristics and functionalities, specified as methods/procedures. Inheritance within a class hierarchy is a prime notion. However the OO model does not allow object methods to be organised in hierarchy (Borgida and Greenspan, 1980). Hence those techniques for modelling software requirements have limitations. It is argued that, in using the standard analysis techniques as indicated above, the analyst must restrict themselves to specifying only high level characteristics of the software, and should not attempt to produce design specification. Techniques such as the entity relationship analysis, which is used during database structure design, can lead the analyst into specifying design aspects of the software rather than specifying the software requirements.

5.2 Formal specification languages

Formal methods of specification of software requirements consist of textual or sometimes tabular and graphical specification languages.

JRDL: The Japanese requirements description language (JRDL) is proposed by Ohnishi, Agusa and Ohno (1985). JRDL is used to express requirements as distinct statements, consisting of single sentences. JRDL statements can be transformed into CRD (conceptual requirement description). CRD consists of primitives consistent with the requirement domain.

RML: The requirements modelling language (RML) was initially proposed as a language specifically designed for specification requirements (Greenspan, Mylopoulos and Borgida, 1994). RML attempts to model domain knowledge by providing an object-centred modelling framework. In RML objects are grouped in classes to which properties can be associated. RML supports activity, entity and relationship objects. RML has been extended into CML (conceptual modelling language) of which TELOS is a version. TELOS is an extensible language in the sense that it allows new object classes to be defined, and where all classes can be defined as subclasses of the class meta-class. For a detailed discussion of TELOS, the reader is referred to Koubarakis, Mylopoulos, Stanley and Borgida (1989) and Mylopoulos, Borgida, Jarke and Koubarakis (1990).

5.3 Specifying non-functional requirements

Mylopoulos, Chang and Nixon (1992) discuss methods for representing system requirements and in particular non-functional requirements. They proposed a process-oriented approach where requirements are expressed in terms of a set of goals, a set of links relating the goals, methods for refining the goals, a set of correlation rules to describe the interactions between the goals, and a labelling procedure. The requirements are expressed using a predicate-like style, eg, a requirement consisting of a goal where attributes of data of type employee are to be recorded accurately, would be expressed as: Accuracy(attributes(employee)).

A corresponding goal refinement method can be expressed as:

Accuracy(attributes(employee)) -> Accuracy(attributes(Secretary)),
Accuracy(attributes(Accountant), ...

since Secretary, Accountants, etc ... are categories of employee (and are described with a different set of attributes).

5.4 Standards for requirement specification

It is worth mentioning that there are a number of requirement specification document standards that have been proposed (Dorfman and Thayer, 1990). These standards can be considered as providing a template of what is to be included in a SRSD.

The IEEE standard 830-1984 comprehensively describes what a software requirement specification document (SRSD) should contain. In essence it describes:

Attributes of a good SRSD:
- unambiguous, complete, consistent etc

Methods for expressing requirements:
- input-output specifications
- use of representative examples
- models: mathematical, functional etc.

Annotation of software requirement

The standard also proposes a prototype outline for an SRSD.

The British Standards Institution also proposes standard (BS 6719:1986) for specification of computer based (software) systems. The British standard proposes a 2 part specification: a summary and a detail description of user requirements.

The summary specification emphasises on environment issues, essential features and time and cost constraints.

The detail specification describes the input and output data characteristics, data manipulation, performance monitoring and acceptance criteria.

5.5 General comments

The purpose of using appropriate specification methods lies in providing a representation of the requirements in a form that facilitates recording, communication and analysis. Traditional representation methods for software artefacts have been used for this purpose, as described earlier. Such methods are successful in representing certain aspects of software requirements, namely those that relate to the software being produced, eg, data entities, processes, etc. However recent development attempt to tackle software representation from a more abstract level, meta-level or extensible level, in the sense that the representation can accommodate new concepts as they are required. Languages such as RML and TELOS fall in this category. Representation of domain knowledge is one of the issues that need further attention. For example, as discussed in section 4.5, 4.6 and later in 6, domain knowledge is necessary in determining the characteristics of software requirements. An extensible representation method can guarantee that additional concept can be introduced into the initial framework. Such representation, however, must ensure that sufficient domain knowledge will be represented.

If software requirements are viewed as evolutionary, rather than representing a final document for a specific purpose, then again an extensible representation method seems highly desirable. In this sense it may be more appropriate to regard (software requirement specification as a highest level software representation and) SRE as a highest level meta process, rather than a temporal phase in the software development process. Then traceability between information of requirements become crucial, as the specification document can then be regarded as a validation document. Representation methods for software requirements must therefore facilitate such traceability.

This ultimately suggests that the very purpose and success of SRE relies heavily on the representation methods used, and the fact that SRE has to deal with informal issues, does not make any currently used method an obvious choice, except for extensible methods.

6 Universe of Discourse (UoD) and Application Domain Knowledge

A key issue to the success of the SRE process is mutual agreement between the user and the software developer on the requirements. Agreement means that the interpretations of the requirement, analysed and expressed by informal, semi-formal or formal methods, by the user and analyst are the same. However in real life situations, users are often unfamiliar with semi-formal or formal methods of requirement analysis and representation. Hence for the purpose of reaching agreement, software developers often have to rely on informal methods. The users interpret the requirement within their universe of discourse(UoD): UoD_u, within the environment they work in and supported by their background experiences and knowledge. Similarly the analysts will interpret the requirement within their UoD: UoD_a. In the best case user and analyst is the same person:

$$UoD\_u = UoD\_a.$$

In general though, the users' and analysts' UoD's will be different. In the most trivial case: the language of communication is part of the users' and analysts' respective UoD's. If they

do not speak the same language, reaching agreement between them will be difficult! But more importantly, the UoD's also include domain knowledge, which provides a framework of concepts and ideas, for reference and interpretation.

The application domain knowledge resides mostly within the users' UoD's. Unless the Analyst is expert in the application area, their knowledge of the application domain is likely to be limited. Since interpretation of the software requirements is performed within the users' and analysts' UoD's separately, user and analyst may place different meanings to the requirements. The greater the difference between the UoD's of the user and analyst, the less likely are their interpretations of the requirements going to be the same.

Assuming that the UoD's of the user and analyst are different, then agreement must be negotiated, ie, user and analyst must attempt to reconciliate concepts between their respective UoD's. This will result into a negotiated common UoD being established. Such negotiation requires representation of the concepts in a language and the negotiated UoD only exists as such, ie, the representation is the negotiated common UoD. A particular concept will have to be interpreted in both participants' initial UoD's and in the negotiated common UoD. Hence the negotiated UoD would ideally have to include sufficient domain knowledge to ensure correct interpretation of the concepts.

Our description of agreement with a negotiated UoD also provides a definition for an assumption, ie: an assumption can be regarded as an un-negotiated concept (forcedly) introduced into the negotiated UoD, and having a parent concept in only one participant's UoD. Assumptions therefore cannot be guaranteed to have matching concepts in both participants' initial UoD's.

Aspects of interpretation of requirements that are likely to differ between user and analyst are:

1. Perception of requirement by user
   Assumptions about user environment
   "Self in-evidence"
   Motivation for requirement
   Rationale for requirement

These matters can be related to a set of concepts which exist prior to the software requirements being elicited.

2. Impact of requirement on user organisation
   Assumptions about user environment
   Perception of requirement by user
   Assumptions about default actions

These matters relate to another set of concepts which exist as a result of the software requirements having been elicited. They concern the user organisation, and can therefore be viewed as pertaining to a larger organisation UoD, UoD_org.

Previously discussed examples illustrate some of these aspects of requirements specification. The above list is not meant to be exhaustive, as other types of "unstated" requirements may be uncovered for particular application domains.

In the SRE process, it is therefore important to acquire domain knowledge, in addition to specific software requirement information, such as functional requirements. But it can be difficult to decide how much domain knowledge is necessary and how much is superfluous. It is equally important to establish and record the relationships between domain knowledge items and related requirements specification.

6.1 Viewpoint approach (CORE)

CORE (COntrolled Requirement Expression) is a method of requirement analysis based on the notion of Viewpoints. The method was first proposed by Mullery (1979) and is recently reviewed in Galley and Smith(1993). The CORE method proposes a "viewpoint" approach to requirements analysis. A system can be analysed in terms of viewpoints, each corresponding to a local context centred on an object, process or state. For example in a library system, viewpoints can be specified for the borrower, the librarian, a book, a book loan, etc. The CORE method also describes a systematic process for establishing and building viewpoints, building a viewpoint structure, validating viewpoints, and combining viewpoints.

Mullery also discusses software tools for supporting his method. A viewpoint can be considered to be a description of the software and requirements based on a *restricted UoD*. For example the software can be viewed from the following perspectives:

User interface and interface with other systems
Operation (how commands are issued by users)
A particular user's perception of the software (as compared to that of another user)
A particular component software within the system
A particular object within the system (eg, a book in a library system)
Etc.

Viewpoints can overlap (eg, the user interface and operation viewpoints will have many properties in common).

The CORE method prescribes a series of steps, analytical tools and representational methods (tables, diagrams) to specify software requirements. The main activities are that of identifying viewpoints, analysing and confirming viewpoints, and combining viewpoints.

In analysing software requirements in terms of viewpoints, the analyst is required to interpret requirement information within several (smaller) UoD's. Issues such as requirement conflicts, incompleteness, ambiguities are more likely to come to light. At the same time the analyst is able to enhance their knowledge of the application domain.

The notion of viewpoints is taken up by Easterbrook(1993), where he discusses domain knowledge modelling. He also discusses conflict resolution method in requirements engineering based on viewpoints hierarchy.

Nuseibeh, Kramer and Finkelstein(1993) discuss relationship between requirements in terms of multiple viewpoints. Viewpoints are described using viewpoint templates, which when completed, instantiate a particular viewpoint. Such a description facilitates the reuse of specification, and a particularly configured set of viewpoints can be used to specify a particular problem domain. A viewpoint template consists of a style slot, a workplan slot, a domain slot, a specification slot, and a work record slot. Relationships between viewpoints are expressed as a set of "inter-viewpoint communication rules" that are invoked. A computer based support environment, known as "The Viewer" is also described. It supports viewpoint definition, rule definition and rule invocation.

7 Supporting the Requirements Engineering Process

Techniques to support the SRE process aim primarily at minimising the effects of the problems in requirements discussed above. Those support mechanisms ranged from formal requirements specification methods, systematic and comprehensive analysis procedures (also discussed earlier) and document management techniques, eg, hypertext, to manage the potentially large amount of information that can be generated. The support methods are often complementary. Formal techniques can be used to resolve problems of requirement specification such as ambiguities, but they are not always understandable by the general (non-technical) user community. Some proposed support mechanisms for SRE are described below.

7.1 The REMAP approach

Ramesh and Dhar(1992) discuss support mechanism for requirements engineering, based on a conceptual model of the SRE process. A prototype support environment REMAP, is also proposed based on the high level conceptual modelling language TELOS, discussed earlier. REMAP supports the capture of design rationale, domain knowledge acquisition and reuse of design knowledge. It also provides for design decision support, maintenance of requirement changes. REMAP is based on a conceptual model of SRE, that views the SRE process as consisting of a set of primitive components. The model is based on an extension of the gIBIS model (Conklin and Begeman, 1988). A requirement is considered to generate a set of interrelated information: issues, arguments, assumptions, position, decision statements and constraints. The emphasis in the above model is to capture the context within which the requirement is being engineered.

7.2 Hypertext based support for SRE

Hypertext based techniques to support the requirement engineering process is considered by Potts, Takahashi and Anton (1994). Their support approach is based on the Inquiry model for requirements analysis. Their discussion is similar to that presented by Ramesh and Dhar. However their approach emphasises the processes compared to that of Ramesh and Dhar who discuss the issue in terms of information components and states. The inquiry cycle is appropriate for hypertext based support. Relationships between requirement elements can be specified by hypertext links. Similarly links to domain knowledge, assumptions, argument information and history of the requirement can be specified to build a hyper-document for requirement specification.

7.3 CORE

The CORE method was introduced earlier (section 5.1) in terms of its ability to handle the UoD. The CORE method for requirement analysis is discussed extensively in the current literature and is an example of a methodology that attempts to support SRE, in particular in assisting the software developer in identifying and understanding the characteristics of and relationships between software requirements.

8 Conclusion

In this paper we have discussed a wide range of issues that impact on the requirements engineering process. A number of specific problems have been described and exemplified. A distinctive characteristic of SRE, also discussed by McDermid(1994), is the so called "*soft*" nature of requirements information, specially when dealing with non-functional requirements. Software requirements relate to matters that are often difficult to formalise and to represent adequately using current representation methods. McDermid also discusses the inadequacies of the "orthodox" specification approach. He argues that requirement specification has to be addressed from a different perspective and suggests a causality and objective oriented approach. A key thrust in McDermid's argument is that software requirements has to be considered as a whole. This means that since software requirements specification relates to the users' needs and perceptions of the behaviour of the software system, they have to relate to their use of the final product. This is contrary to the idea that the main purpose of an SRSD is for use as a reference document for later design and construction of the software system.

In support of McDermid's position, the argument put forward in this paper is that SRE has to be treated as a cohesive integrated process, the product of which is not only a set of functional and non-functional specifications describing the software behaviour, but also a

representation of concepts and relations between them, that captures the characteristics of the user world within which the software is to be embedded. This argument follows from a careful observation of the SRE process as it occurs. Although it is convenient to describe SRE as consisting of three activities: elicitation, analysis and specification, neither activity can be regarded as taking place on its own at distinct periods of time. The process of requirement elicitation occurs primarily within the user environment, and involves the user as the main source of information. It is highly interactive, where ideas are continuously put forward, debated, modified and refined. The requirement information thus identified must be recorded and represented in some form. Requirement analysis brings to light the characteristics of those requirements and has to be performed in the proper context that recognises the characteristics of the application domain. Underlying both processes are representational and specification methods that provide a basis for communication and agreement between participants. We consider the representation document for the software requirement as forming a negotiated common UoD between all participants. Furthermore the specification document can be regarded as a contractual document between users and software developers. However to seal the software requirements into such a specification document is futile, since software requirements are inherently dynamic and subject to constant evolution. Hence the volatility of software requirements must be accounted for during SRE.

Software requirements need to be validated and verified. Validation has to be a user centred process, allowing the users to ultimately confirm that their expectations of the software behaviour have been satisfied. Although techniques such as prototyping can be used for such user validation, the real test would be acceptance of the final software product. This suggests that the SRE process cannot terminate until the final product has been delivered. Hence SRE must be more appropriately regarded as a pervasive process, the aim of which is to guide the complete software development process, in providing the purpose for each activity of software production.

The requirements for providing support for the SRE process need to be identified. Support methods should desirably take into consideration the initial informal or "'soft" nature of the information elicitation process, allowing for requirements and domain information to be recorded adequately. We require a more effective approach of recording information, and means of establishing their importance and relevance to the software requirements. Extensible representational methods must be available to capture the concepts upon which the user world is constructed, with particular emphasis on depicting the relationships between those concepts. Validation of software requirements requires effective traceability mechanism, ie, ability to trace requirement information from initial elicitation, throughout its evolution and up to final usage of the software. Development of software tool for such support role hinges on coherent and adequate knowledge acquisition, representation and management methods.

References

Borgida A. and Greenspan S..J. (1980), "Data and Activities: Exploiting Hierarchies of Classes", in *Proc of Workshop on data Abstraction, Databases and Conceptual Modelling,* M. Brodie and S. Zilles (eds), Joint SIGART,SIGMOD, SIGPLAN Newsletter.

Conklin J. and Begeman M. (1988), "gIBIS, a Hypertext tool for exploratory policy discussion", in *ACM Transaction on Office Information Systems*, vol 6, pp303-331.

Davis A.M. (1990), "*Software Requirements, Analysis and Specifications*", Englewood Cliffs, N.J., Prentice Hall.

Dorfman M. and Thayer R. (1990), "*Standards, Guidelines and Examples on System and Software Requirements Engineering*", IEEE Computer Society Press Tutorial, IEEE CS Press.

Easterbrook S. (1993), "Domain Modelling with Hierarchies of Alternative Viewpoints" in *IEEE Intl. Symposium on Req. Eng.: RE '93.* IEEE CS Press pp65-72.

Finkelstein A. (1994), "Requirements Engineering: a review and research agenda", in *Proceedings of Asia Pacific Conference on Software Engineering, IEEE CS Press. pp10-18.*

Galley D. and Smith T. (1993) "Overview of CORE Techniques", in *Software Engineering: A European Perspective,* IEEE CS Press, pp97-104.

Goguen J.A. and Linde C. (1993), "Techniques for Requirements Engineering" in *IEEE International Symposium on Requirements Engineering, RE '93,* IEEE Computer Society Press, pp 152-164.

Goguen J.A. (1994), "Requirements Engineering as the reconciliation of social and technical issues", in M Jirotka and J Goguen, Requirements Engineering: Social and Technical Issues, Academic Press, pp165-200.

Gotel O.C.Z and Finkelstein C.W. (1994), "An Analysis of Requirements Traceability Problem", in *Proc. of the 1st Conf. on Requirements Engineering,* IEEE CS Press, pp94-101.

IEEE (1977), *Transaction on Software Engineering*, 3, 1, Special edition on requirements engineering. IEEE CS Press, 1977

IEEE (Institute of Electrical and Electronics Engineers) (1983) *IEEE Glossary of Software Engineering Terminology,* ANSI/IEEE Standard 729-1983.

Koubarakis M., Mylopoulos J., Stanley M. and Borgida A. (1989), "Telos: Features and Formalization", *Technical reports on Knowledge Representation and Reasoning: KRR-TR-89-4,* Dept of Computer Science, Uni. of Toronto, Ontario, Canada.

Kramer J. and Ng K. (1988) "Animation of Requirements Specfications", in *Software-Practice and Experience*, vol 18, no 8, pp749-774.

Lubars M., Potts C. and Richter C. (1993), "A Review of the State of Practice of Requirements Modelling", in *IEEE International Symposium on Requirements Engineering, RE '93,* IEEE Computer Society Press, pp 2-14.

McDermid J.A. (1994), "Requirements analysis: Orthodoxy, fundamentalism and heresy" in M Jirotka and J Goguen (eds), Requirements Engineering: Social and Technical Issues, Academic Press, pp17-40.

Mullery G.P (1979). "CORE- A Method for Controlled Requirement Engineering", in *Proc. 4th Intl Conf. on Software Engineering,* IEEE CS Press, pp126-135.

Mylopoulos J., Chang L. and Nixon B. (1992), "Representing and Using NonFunctional Requirements: A Process-Oriented Approach", in *IEEE Trans. on Software Engineering,* 18, 6, 483-497.

Nuseibeh B., Kramer J. and Finkelstein A. (1993), "Expressing the relationship between Multiple Views in requirements Specification" in Proc. 15 International Conference on Software Engineering, IEEE CS Press, pp187-196.

Ohnishi A., Agusa K. and Ohno Y. (1985), "Requirements Model and Method of Requirements Definition", in *Proc. IEEE COMPSAC '85,* IEEE CS Press, pp26-32.

Potts C, Takahashi K. and Anton A. (1994), "Inquiry-Based Requirements Analysis", in *IEEE Software,* March 1994, pp21-32.

Ramesh B. and Dhar V. (1992), "Supporting Systems Development by Capturing Deliberations Requirements Engineering", in *IEEE Trans. on Software Engineering,* 18, 6, pp498-510

Soares J.O.P. (1994), "Underlying concepts in Process Specifications", in *Proc. of the 1st Conf. on Requirements Engineering,* IEEE CS Press, pp48-52.

Sommerville I., Rodden T., Sawyer P., Bentley R. and Twidale M (1993), "Integrating Ethnography into the Requirement Engineering Process", in *IEEE International Symposium on Requirements Engineering, RE '93,* IEEE Computer Society Press, pp 165-173.