Combining Knowledge and Metrics to Control Software Quality Factors

Jordi Alvarez, Núria Castell, and Olga Slavkova

Abstract

The LESD project (Linguistic Engineering for Software Development) aimed to develop computing tools for analysis and reasoning on functional or preliminary specifications of aerospace software written in English. These tools help to control the quality of software written during the first stage: specification. The factors considered relevant to the quality of specifications in the LESD project are: traceability, modifiability, completeness, consistency, and verifiability. This paper deals with completeness and modifiability. In the case of completeness we present a symbolic approach to control this factor, using a Knowledge Base. Checks are based on *metarequirements* that try to ensure structural completeness. The concept of modifiability is based on the level of interconnection between the requirements of the specifications. Two metrics have been defined in order to measure global and local levels of interconnection.

1 introduction

The work presented in this paper is a continuation of a project (LESD) carried out in collaboration between French and Spanish researchers. The LESD project (Linguistic Engineering for Software Development) (Borillo et al.,1991) was initiated in the ARAMIIHS centre, Toulouse (France). Researchers taking part were drawn from IRIT–CNRS (Institut de Recherche en Informatique de Toulouse), the Université Paul Sabatier, the Université de Le Mirail, MATRA MARCONI SPACE, and from the Universitat Politècnica de Catalunya (under the terms of a joint Franco–Spanish initiative, maintained during 3 years). The aim of the LESD was to develop computing tools for the analysis and reasoning employed in drawing up functional or preliminary specifications for aerospace software written in English. These tools help to control the quality of software written during the first stage: specification. LESD selected this stage because, as stated in (Pressman,1992), it is really important to detect errors as soon as possible. The quality control is based on two aspects: the writing norms and the software quality factors. In this paper we only deal with the second aspect.

The LESD architecture (Castell et al.,1994) comprises two parts: the first consists of the syntactic–semantic and the domain analysis of the specifications, from which a conceptual representation of such specifications is obtained; the second takes in reasoning mechanisms relating to the representation of requirements. Therefore LESD falls in the fields of Linguistic Engineering and Knowledge Based Systems.

The work carried out so far on the LESD project has consisted of developing syntactic and semantic analysis tools for such specifications, a study of the knowledge required to interpret those specifications, the design of a suitable knowledge representation system (using a frame–based formalism) and the implementation of reasoning mechanisms for evaluating the quality factors in the specifications at a symbolic level, within the space field. The requirements making up a specification are successively analyzed and interpreted and subsequently incorporated in the Requirements Base while taking the domain representation (Knowledge Base) into account.

The typology of the objects and activities of the domain were defined, as were the relationships for structuring the lexicon and the entities of the domain: taxonomic relationships (the is–a relationship), meronomic relationships (decomposition of an object into its components), temporal relationships (particularly between activities), characterization (for example, status characterizes system) and thematic functionality (agent, object, etc).

In addition to symbolic control of quality, measurement algorithms are currently being added for measuring the five quality factors considered in LESD: traceability, modifiability, completeness, consistency, and verifiability. Evaluation of these factors at a symbolic level requires the development of reasoning algorithms applied to the conceptual representation of specifications expressed in natural language.

The traceability factor has been developed to date, this being of particular importance in software design (Borillo et al.,1992). In defining this factor in LESD, an interactive approach has been adopted in which the engineer investigates by specifying a set of entities which are interconnected by relationships. The system responds by providing a list of requirements whose conceptual representations contain these entities and relationships. The algorithm developed for analyzing enquiries and calculating responses is based on the notion of type. Calculation of the responses involves activating an inference mechanism operating on the Knowledge Base. A detailed description may be found in (Toussaint,1992).

At present, the Spanish group works in the research subjects left open by LESD. We are studying the other selected quality factors, following the mentioned knowledge–based approach and adding a metrics–based approach. In particular we are dealing with the control of completeness factor at symbolic level (Álvarez et al.,1994) and with metrics to measure the modifiability factor (Castell et al.,1995b). The work related to these two factors is explained in this paper. A hierarchical model of software quality control based on software measurement has been developed and applied to the traceability and the modifiability factors (Castell et al.,1995c). Also we are developing an assistance system for writing software specifications in natural language (Castell et al.,1995a).

This paper is organized in two main sections. Section 2 is devoted to the completeness factor: definition, kinds of completeness, and symbolic control of this factor. Section 3 is devoted to the modifiability factor: its definition, definition of adequate measurements, and experimental work.

2 The Completeness Factor

The problem of generating complete specifications is crucial to software development life cycle. Incomplete specifications are a big source of misunderstandings between the client and the software engineer. When there are some pieces of information that are needed but missing from the specification, the persons who take information from it (designers, programmers, formal specification writers or anyone else; hereafter called specification–readers) tend to fill these holes with their own surmises. As specification–writers have generally a different point of view from specification–readers, these surmises differ at least a little from what specification–writers took for granted when writing the specification. This would cause the final system behaviour being different from the one that was initially thought. Therefor these gaps in the specification can result in problematic design and code changes. Checking automatically whether a specification is complete or not will avoid correcting errors in later stages of the software development process.

2.1 Problem Definition

To get inside the problem we will start by defining it. The (IEEE,1984) standard states that *a specification is complete when all the requirements relative to functionality, performance, constraints on system structure, attributes and external interfaces are written and if all the terms used in these requirements are defined.*

This definition is too concrete and says too many things to reason in an abstract manner about it. We prefer to give our own definition. But before doing that, we must first remember that a specification is a document to allow information interchange between different persons involved in a software engineering project. That is, the purpose of the specification is to ensure conceptual completeness among software engineers. Then, if we assume that the aim of this software engineering project is to construct a system to achieve some goals, we can say *a specification is complete if all the information needed to construct the specified system is stated.*

To deepen into the given definition, we must state several questions. The first one, *which information do we need in order to construct a system?* This question brings up the problem of deciding which pieces of information are relevant. Answering it establishes a link between the needed information and the purpose of the specification. As the system is specified through the specification, the information a specification needs in order to be considered complete depends on the specification itself.

By other hand, the action of constructing the system implies that the persons who will construct it know the exact expected behavior of the system. So, this definition implies the specification contains all the information needed to know the exact behavior of the system. This raises another question: *what do we understand as exact behavior?* The answer relates to the kind of specification we are dealing with. If it is a preliminary specification (as it is the case), we cannot expect every detail of the system to be present in it. The more detailed is considered to be the specification, the more information we must check to be present in it.

Finally, as the specification will be read by human persons and these persons will have a background and common–sense knowledge, a third question must be answered: *which part of the relevant information must be stated explicitly in the specification and which part can be taken for granted?* Background and common–sense knowledge makes needed information a subset of relevant information.

More precisely, taking into account the (ESA,1991) and (IEEE,1984) standards, and the paper (Cordes et al.,1989), the characteristics a specification must have in order to be complete are:

1. All system relevant characteristics must be present in the specification. No mind they relate functionality, performance, design constraints...

2. Every object that is referenced in the specification must also be defined. This feature could be thought as a special case of (1), so the mentioned standards do not specify it.

3. No information is left unstated or to be determined. The use of TBDs (to be determined) must be avoided.

4. The system response must be specified for all realisable classes of input data in all realisable classes of situations.

5. All figures, tables and diagrams in the document must be full labelled and referenced.

The first three characteristics constitute structural completeness, the fourth one refers to logical completeness and the last one defines documental completeness. In this paper we deal with structural completeness. For a general discussion about different kinds of completeness see (Tuells et al.,1993).

We must consider two kinds of structural completeness: external and internal. Point 1 is referred as external completeness so we have the hint to the incompleteness in the system, external to the specification itself. Points 2 and 3 are referred as internal completeness so we get the pointer to the incompleteness in the specification itself.

2.2 External completeness

Obviously, external completeness is harder to detect than internal one. The reason is that we only have the specification to detect missing information from the specification itself. As has been said in the previous section, we would find pointers to this missing information in the system; but the only knowledge we have about the system comes from the specification. We can try to supply this lack of information with an extensive domain Knowledge Base that can give some hints about possible incompletenesses. In this way, we can use this domain Knowledge Base to convert some external incompletenesses to internal ones; i.e. using *metarequirements* as explained in the next section.

If we have no domain knowledge, we can check whether an object is defined, but no more can be done. As stated in (Reubenstein et al.,1991), there is no way to detect the absence of information that is orthogonal to the knowledge we have about the system. The more domain knowledge we have, the less incompletenesses will be orthogonal to the knowledge we have.

For example, if we are specifying a library management system, the library could have been defined as a repository of books. We could know from our domain knowledge that an element of a repository must have a unique identifier. As our system manages a library, books in it will be elements of a repository and must have a unique identifier. Therefore we must check the definition of book for a unique identifier. If we do not find it, we can assure that the definition of book is incomplete for this problem.

Of course, we cannot guarantee that every incompleteness will be catch. So, as stated in (Reubenstein et al.,1991), the end–user has to be the final arbiter of completeness.

2.3 Internal completeness

The internal completeness ensures that all the information present in the document is completely defined. Cordes and Carver (1989) propose a simple algorithm to check this kind of completeness. What they do is mainly to check a minimum set of properties for each object and event that appears in the specification. In the Requirements Apprentice (Reubenstein et al.,1991) we also see this minimum set of properties: when instances are linked to clichés, a set of expectations in the form of roles that must be filled is generated. It also maintains a list of undefined things.

In addition to this minimum set of properties, we propose checking some properties resulting from the reasoning about the conjunction of the domain knowledge and the overall knowledge we have about the specification ; i.e. the Requirements Base. These properties are intended to express specification quality factors.

Our initial idea is to establish the set of properties that must be checked through the use of *metarequirements*: requirements about the knowledge we have about the system; that is, about the specification itself[1]. These *metarequirements* specify the quality properties we are talking about in the same way system requirements specify an activity. That is, the domain knowledge can contain some requirements that must be checked over the specifications. As a result of this, we will have requirements that refer to the system and requirements that refer to the specification. In this sense, the set of *metarequirements* could be seen as the *standard* the program follows to check the structural completeness or to check another quality factors if they are defined.

No distinction will be made between requirements and *metarequirements*. This way, the specification can also contain *metarequirements* conditioning its completeness. For example, the requirement "Every system needs an I/O device in order to be controlled" is, in fact, a *metarequirement* that forces the definition of controlled systems to have an I/O device.

This is quite easy. More interesting is the possibility to deduce *metarequirements* from system requirements. For example, if we are talking about an emergency system and we read the requirement "Each audio emergency signal shall have a tone specific to each condition", we will know that every audio emergency signal defined may have some conditions defined and each one of these conditions must have a specific tone[2].

As we have seen earlier, it is very difficult to find incompletenesses. Only a few incompletenesses will be noticed by the user if we only report those we are sure about. Instead, a better approach can be to report also some possible incompletenesses. The modality of a requirement, used in (Toussaint,1992) to give an idea about the importance of the requirements (needed, desirable, in future plans...), can be used to do this work. So, a *metarequirement* can tell us about a definite incompleteness or can warn about a possible one.

2.4 Computing internal completeness

A first thought to compute internal completeness can be to check every completeness property expressed through a *metarequirement*. The idea is similar to that of metarelationships in the KAOS system (Lamsweerde et al.,1995), but applied specifically to completeness specification validation.

We have requirements referring the specification (that we call *metarequirements*) and requirements referring the system we are specifying. Completeness properties will always be specified in *metarequirements*, but these properties can be influenced by any kind of requirement in the specification. In this way, system requirements can also participate in completeness checking.

In the library example, we will check for the book unique identifier (*metarequirement*) whenever the library management system performs any operation

[1] We call them *metarequirements* because in some way they are requirements over the requirements that define the system we are specifying.

[2] Although the specificity of the tone relates to consistency, its presence is a completeness issue.

that requires books having a unique identifier. When there is no operation that requires books having a unique identifier, no check must be done.

The completeness check for the overall specification can be computed easily checking the completeness for each entity referenced in the specification. Looking at the Knowledge Base as a whole, the specification will be complete if it provides enough knowledge for the requirements activities to be performed. Looking at each entity, we can say its definition is complete in the context of the specified system if it provides enough knowledge for any related requirement activity to be performed.

Completeness properties constitute the knowledge we will use to check if there is enough knowledge to perform an activity. As these properties will refer to one or several entities, the overall specification completeness can be deduced from completeness checks over all entities related to it. Going on with this idea, there are two ways we can check for incompletenesses concerning an entity: in a *static* way and in an *operational* way.

The first way consists in checking every property related to an entity that must also be accomplished in order to allow every requirement in the specification to be feasible. For example, going on with our library management system, the static completeness check for book would result in looking for every related property. We would find a property saying: "Repository elements must have a unique identifier". As the specified system must manage a library, and a library is a repository of books, we know that books are repository elements. The next step is deciding whether the specified system needs this property to be accomplished. So, we search any activity that needs that property to be accomplished. We would find for example a reference to the operation of checking out repository elements. Then, as the library management system must be able to check out books (we have a system requirement specifying it) and checking out books is a special case of checking out repository elements (because in out problem, books are repository elements), books must have a unique identifier. So, this *metarequirement* is checked against book definition.

The second way is more ingenious, and is based on checking that an entity is able to perform any action it needs to perform. For example, we want to check operatively our library management system. We would find (among others) a requirement saying that the library management system must check out books. The activity checking out books is more specific than checking out repository elements, and this activity needs repository elements having a unique identifier. So, we must check book definition for unique identifier.

Checking completeness in a static and operational way can be very useful sometimes, but not for an interactive system that wants to check the requirements completeness when it receives them. If we check the completeness for every referenced entity in the requirement we would be repeating a lot of checks.

It is important to note that the checks that are carried out for a requirement depend on both the requirement and the Requirements and Knowledge Bases we have in the moment we incorporate to it. By other hand, it must be noticed that the introduction of a new requirement into the Requirements Base can result in the combination of three different situations that require only specific completeness checks for each one. Taking advantage of this fact, we can manage to perform all checks only once. The skeleton of the algorithm is as follows, for a more detailed explanation see (Álvarez et al.,1994):

1. Retrieve the main activity associated to the requirement (it can be either a system requirement or a *metarequirement*).
2. Perform the following checks over the activity:

   – Static: if the activity represents a property (so, the requirement is in fact a *metarequirement*) and it is needed to perform another activity, the property is checked.

   – Operational: check if the activity can be done (this implies checking properties that condition the activity).
3. For any new entity referenced, perform all operational and static completeness checks.

About completeness properties, by the moment, the implemented prototype only treats two different properties: to have an attribute and to be a concept. The first one checks the entity definition for the corresponding attribute and the second one checks the entity to be an instance of a specified class (or a subclass of it).

3 The Modifiability Factor

The approach to specifications modifiability in LESD has to address two tasks: first, analyze the level of complexity in the modifications with regard to both the requirements taken as a whole (global measurement) and for each individual requirement (local measurement) and, second, select the list of requirements which may be affected by a given modification. Automating the approach to the modifiability of specifications allows both the global and local evaluation objectives to be achieved and avoids missings in the list of requirements to be reviewed as a result of an implemented modification.

3.1 Definition

In (IEEE,1984) the modifiability of specifications is defined in relation to the level of redundancy involved and the simplicity, completeness, and consistency of the modifications. The redundancy involved and the simplicity of carrying out modifications within a set of specifications in LESD are characterised by the level of interconnection between the requirements of the LESD specifications. The intuitive idea is evident: the greater the level of modification, the greater the difficulty of making the modification and the greater the possibility of detecting redundancy in the requirements. The complexity and consistency of the modifications depend on the level of propagation of a given modification in all requirements affected by that modification. Thus we have formalized the basis of the concept of modifiability in LESD in function of the level of interconnection between the specifications requirements. The interconnection between different requirements is defined by common use of entities defined in the LESD domain (i.e. by common information).

The measurement model of modifiability, in addition to indicate the global modifiability of the requirements taken as a whole, must be appropriate for defining the local level of modifiability in case of a specific modification and indicate the subset of requirements which may require modification as a result of such an alteration.

The most suitable mathematical model for calculating the quantity of the common information in a set of requirements expressed through entities (defined in the domain of LESD) in common use, is described in (Emden,1970). The model has been adapted for calculating the level of complexity of a program in (Robillard et al.,1989). In this model the interconnections in a set of predicates via objects in common use are represented by an interconnection table defined as follows:

$$table(object_i, predicate_j) = \begin{cases} X, & \text{if the predicate contains the object,} \\ 0, & \text{if this is not the case.} \end{cases} \tag{1}$$

In the next subsections we describe how looks our interconnection table and the new metrics we have defined, following Emden's model, to measure the modifiability factor.

3.2 The Interconnection Table

We have defined the interconnection table for a set of specifications requirements as follows: each row in the table corresponds to the identification of a requirement and the columns correspond to the various LESD domain entities (i.e., objects, activities, temporal relationships). The interconnection table is obtained automatically using a reasoning algorithm applied to the Requirements Base and its construction is carried out by analyzing each of the requirements in turn.

The $Table(row_i, column_j)$ values may be 0, 1, $\neg 1$, 2, $\neg 2$ or 3 according to the level of entity$_j$–requirement$_i$ dependence and its type (asserted or negated). A detailed explanation of the process to construct the interconnection table, as well as the process to define a m–partition of a set of requirements (m mutually independent subsets), can be found in (Castell et al.,1995b,1995c)

3.3 Global Measurement of Modifiability

The amount of common information in a set of elements is an indicator of the level of interconnection of the elements within this set and is called in (Emden,1970) $excess-entropy$ defined as a difference between $entropies$. The formulas for calculating the entropy H and the excess–entropy C proposed in (Emden,1970) for an m–partition of a set of n elements, and the entropy H_i of a partition$_i$, are:

$$H = \log_2 n \;-\; \frac{1}{n}\sum_{i=1}^{m} n_i \log_2 n_i \tag{2}$$

$$C = \sum_{i=1}^{m} H_i \;-\; H \tag{3}$$

In order to apply Emden's mathematical model to the set of specifications requirements under LESD, let us consider an m–partition of the set of n requirements in mutually independent subsets of requirements such that n_i requirements of each subset$_i$ (i=1,m) are inter–linked by common entities and

$$\sum_{i=1}^{m} n_i = n \tag{4}$$

Let us associate the set of non negative numbers $\{\frac{n_1}{n}, \ldots, \frac{n_m}{n}\}$ to m partitions;

$$\sum_{i=1}^{m} \frac{n_i}{n} = 1 \tag{5}$$

Clearly each number $\frac{n_i}{n}$ indicates the probability that a requirement belongs to the partition$_i$. We shall now define the link between two requirements in terms of the interconnection table.

Two requirements **i** *and* **j** *are inter–linked if at least one entity* **k** *is present so that Table (i,k) ≠0 and Table(j,k) ≠ 0*

The entropy H_i is calculated considering only n_i requirements in the subset$_i$. In general there may exist a k–partition in the subset$_i$. In this case the measurement of H_i is performed using the following formula:

$$H_i = \log_2 n_i \;-\; \frac{1}{n_i}\sum_{j=1}^{k} n_{ij} \log_2 n_{ij} \tag{6}$$

Should there be no partition in the (sub)set of k requirements, the formulas for calculating the entropy and the excess–entropy will be the same as the calculation formulas proposed by (Robillard et al.,1989):

$$C = \sum_{i=1}^{k} H_i \;-\; H \tag{7}$$

$$H_i = \log_2 n_i \; - \; \frac{1}{n_i} \sum_{j=1}^{l_i} p_j \log_2 p_j, \tag{8}$$

where l_i is the number of different configurations of the rows values in the interconnection table$_i$ and p_i is the number of times the configuration$_i$ is repeated. To calculate l_i and p_i let us consider the following definition:

Two requirements **i** *and* **j** *have the same configuration if the following holds true for all the columns of the subtable corresponding to the subset of requirements: Table(i, column) $\neq 0 \Longleftrightarrow$ Table(j, column) $\neq 0$*

If all the values in the rows coincide in addition to having the same configuration, we can say that there is *redundancy* in the requirements set.

In order to measure the level of interconnection independently of the size of the set of requirements a new *Interconnectivity Level* metric is defined:

$$IL = \frac{C}{C_{max}} \tag{9}$$

The quantity C_{max} represents the maximum excess–entropy of a set of requirements (when all requirements are inter–linked) and thus logically normalizes the quantity C. The range of the values of IL metric will be $[0 \ldots 1]$. 0 indicates that there is not common information between the requirements of a set, and 1 indicates that all requirements are inter–linked.

3.4 Local Measurement of Modifiability

To locally evaluate modifiability of a requirement another metric – *Individual Interconnectivity Level* – is defined which reflects the relationship between the individual interconnectivity level of a requirement$_i$ (C_i) and the level of global interconnectivity (C) of the set.

$$IIL = \frac{C_i}{C} \tag{10}$$

The range of the values of IIL metric will be $[0 \ldots 1]$. The IIL metric values near to 0 indicate that the influence of a requirement on the global modifiability of the set of requirements is small. In calculating C_i, two cases can be distinguished:

– if we are interested in the interconnectivity level of the requirement, considering all its entities (both directly referenced ones and related ones), C_i is calculated on the subtable of the subset to which the requirement$_i$ belongs;

– if we are interested in the interconnectivity level of requirement$_i$, considering a subset of such entities, the subtable is constructed following the same steps but substituting the definition of the link between two requirements:

Two requirements **i** *and* **j** *are inter–linked if for each entity* **k** *of a subset the following conditions are met: Table (i,k) $\neq 0$ and Table(j,k) $\neq 0$*

Table 1: *Interconnectivity level* values

	set_1	set_2	set_3	set_4	set_5
n	16	13	13	13	18
m	2	2	3	2	1
n_1	15	12	9	11	-
n_2	1	1	3	2	-
n_3	-	-	1	-	-
IL	0.055	0.066	0.075	0.079	1

3.5 Experimental work and discussion of results

The measurement model of global modifiability has been tested with real data to demonstrate its sensitivity to the interconnectivity level: an improvement of the modifiability in a set of requirements lowers the metric value. A data set of metric values has been obtained from five sets of requirements. For a detailed description of these sets, the interconnection tables and the results see (Castell et al.,1995b). Table 1 summarizes the obtained results.

In (Kitchenham et al.,1990] a robust statistics method is suggested to describe software data sets, thus we have applied this method to identify the range of metric acceptable values: [0.066, 0.079]. From the view–point of measuring the modifiability, one IL metric value within the range of acceptable values is better in comparison with another IL metric value within the range of acceptable values if it is lower. The applied statistical method suggests a quick review of sets with metric values within [0, 0.066[and]0.079, 0.12], and more stringent review of sets with values within]0.12, 1].

4 Conclusions and Future Work

The work carried out in the LESD project consisted in developing the tools for analyzing specifications written in natural language. Five factors concerning quality of specifications were dealt with (traceability, completeness, consistency, verifiability, and modifiability). The techniques for evaluating traceability had already been developed. Currently work deals with completeness and modifiability.

In the case of completeness we have followed a knowledge–based approach. The control is based on *metarequirements* which are represented in a similar way as requirements. The reasoning mechanism operates on the Knowledge Base and the Requirements Base. As a further work, we will explore the relation between traceability and completeness.

On the other hand, we have develop and implement the modifiability measurement applicable to conceptual representation of specifications. We have formalized the basis of the concept of modifiability in function of the level of interconnection between the specifications requirements. Two metrics have been defined: *Interconnectivity Level* and *Individual Interconnectivity Level*. The measurement model is based on the notion of $excess - entropy$, and a robust statistics method has been used to identify the range of metric acceptable values. In order to validate the proposed model, it is necessary to test empirically whether the modifiability measures are good valuations of the actual time and cost of further modifications of the specifications.

4.0.1 Acknowledgements

This work is partially supported by CICYT Spanish institution (TIC93-420) and by CIRIT Catalan institution (GRQ93-3.015 and a postgraduate grant for the first author).

References

[1] Álvarez, J. and Castell, N. (1994) An Approach to the Control of Completeness Based on MetaKnowledge. *Research Report* LSI–94–50–R. Dept. LSI, Universitat Politècnica de Catalunya, Barcelona, Spain.

[2] Borillo, M., Castell, N., Latour, D., Toussaint, Y. and Verdejo, M.F. (1992) Applying Linguistic Engineering to Software Engineering: The traceability problem, in *Proceedings of 10th European Conference on Artificial Intelligence – ECAI'92* (ed. B. Neumann), John Wiley & Sons.

[3] Borillo, M., Toussaint, Y. and Borillo, A. (1991) Motivations du project LESD, in *Proceedings of Linguistic Engineering Conference'91*, Versailles, France.

[4] Castell, N. and Hernández, A. (1995a) Filtering Software Specifications Written in Natural Language, in *Proceedings of 7th Portuguese Conference on Artificial Intelligence – EPIA'95, Lecture Notes in Artificial Intelligence*, Springer–Verlag.

[5] Castell, N. and Slavkova, O. (1995b) The modifiability Factor in the LESD Project: Definition and Practical Results. *Research Report* LSI–95–7–R. Dept. LSI, Universitat Politècnica de Catalunya, Barcelona, Spain.

[6] Castell, N. and Slavkova, O. (1995c) Metrics for Quality Factors in the LESD Project, in *Proceedings of 5th European Software Engineering Conference – ESEC'95, Lecture Notes in Computer Science*, Springer–Verlag.

[7] Castell, N., Slavkova, O., Tuells, A. and Toussaint, Y. (1994) Quality Control of Software Specifications Written in Natural Language, in *Proceedings of 7th International Conference on Industrial & Engineering Applications*

of Artificial Intelligence & Expert Systems – IEA/AIE'94 (eds. F.D. Anger, R.V. Rodriguez, M. Ali), Gordon and Breach Science Publishers.

[8] Cordes, D.W. and Carver, D.L. (1989) Evaluation method for user requirements documents. *Information and Software Technology*, **31**(4), 181–8.

[9] van Emden, M.H. (1970) Hierarchical Decomposition of Complexity. *Machine Intelligence*, **5**, 361–80.

[10] European Space Agency (1991) ESA software engineering standards, Issue 2, February.

[11] IEEE (1984) Guide to Software Requirements Specifications, ANSI/IEEE Std. 830–1984.

[12] Kitchenham, B.A. and Linkman, S.J. (1990) Design Metrics in Practice. *Information and Software Technology*, **32**(4).

[13] van Lamsweerde, A., Darimon, R. and Massonet, P. (1995) Goal–directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt, in *Proceedings of 2nd International Symposium on Requirements Engineering*, IEEE CS Press.

[14] Pressman R.S. (1992) Software Engineering: A Practitioner's Approach. Mac Graw Hill, New York.

[15] Reubenstein, H.B. and Waters, R.C. (1991). The Requirements Apprentice: Automated Assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, **17** (3), 226–40.

[16] Robillard, P.N. and Boloix, G. (1989) The Interconnectivity Metrics: A New Metric Showing How a Program is Organized. *The Journal of Systems and Software*, **10**, 29–39.

[17] Toussaint, Y. (1992) Méthodes Informatiques et Linguistiques pour l'aide a la Spécification de Logiciel. *Ph.D. Thesis*. Université Paul Sabatier, Toulouse, France.

[18] Tuells, A. and Castell, N. (1993) The Completeness Problem in LESD. *Research Report* LSI–93–26–R. Dept. LSI, Universitat Politècnica de Catalunya, Barcelona, Spain.

biography

Jordi Álvarez is graduate in Computer Science by the Universitat Politécnica de Catalunya (1993). At present he is a Ph.D. student in the Artificial Intelligence Program of the LSI Departament (UPC). His main research interests are knowledge representation, neuronal networks, and machine learning.

Núria Castell is graduate in Computer Science by the Universitat Autònoma de Barcelona (1981) and received her Ph.D. degree in Computer Science from the Universitat Politécnica de Catalunya (1989). At present she is "Profesora Titular de Universidad" (similar to Associate Professor) in the LSI Departament (UPC). Her research interests include knowledge representation, and natural language processing.
Olga Slavkova is graduate in Computer Science by the Universidad Central de las Villas, Cuba (1987). Presently on leave of research. She was "Profesora Asociada" (hired Teacher) in the LSI Departament (UPC) and at the same time Ph.D. student in the Software Program of the LSI Departament. Her main research interest is software metrics.