An Approach to the Control of Completeness
Based on MetaKnowledge

Jordi Alvarez
Núria Castell

Report LSI-94-50-R
AN APPROACH TO THE CONTROL
OF COMPLETENESS BASED ON METAKNOWLEDGE

Jordi Alvarez, Núria Castell
Departament de Llenguatges i Sistemes Informàtics
Pau Gargallo, 5. Barcelona 08028 SPAIN
e-mail: jalvarez@lsi.upc.es
e-mail: castell@lsi.upc.es

ABSTRACT

The work presented in this paper propounds a solution to the problem of controlling the completeness of a set of requirements written in natural language. The work is integrated in LESD project (Linguistic Engineering for Software Development), which aim is to develop a computer aided system for analysing and reasoning about software requirements. These requirements are intended to fit some quality factors (traceability, completeness, modifiability, consistency, and verifiability).

In LESD, the conceptual structure of natural language specifications is represented using a frame-based knowledge base and the checks to ensure completeness are performed over it. These checks are based on metarequirements that try to ensure structural completeness (internal and external). So, our Knowledge Base contains general knowledge about the domain (aerospace in LESD project), the requirements, and the criteria to control the quality factors. A prototype has been developed in order to test this approach.

RESUM

El treball presentat en aquest article proposa una solució al problema del control de la completesa d’un conjunt de requeriments escrits en llenguatge natural. El treball s’integra en el projecte LESD (Linguistic Engineering for Software Development), que té com a objectiu desenvolupar un sistema d’ajuda per l’anàlisi i el raonament sobre requeriments de software. Aquests requeriments han d’ajustar-se a uns factors de qualitat predeterminats (traçabilitat, completesa, modificabilitat, consistència i verificabilitat).

A LESD, l’estructura conceptual de les especificacions està representada usant una base de coneixement de tipus frames, realitzant les comprovacions necessàries sobre ella. Aquestes comprovacions estan basades en metarequeriments que intenten assegurar la completesa estructural (tant interna com externa). Així doncs, la nostra Base de Coneixement conté coneixement general sobre el domini (especialment a LESD), els requeriments i els criteris de control dels factors de qualitat. Un prototipus ha estat desenvolupat per tal de comprovar la viabilitat d’aquesta aproximació.

---

1This research is supported by CICYT and CIRIT Spanish institutions.
1. INTRODUCTION

The work presented here is part of LESD project (Linguistic Engineering for Software development). The first goal of LESD is the development of tools to analyse and reason about preliminary and functional specifications written in natural language. In particular, we are working with specifications about aerospace systems, written in English.

A specification in LESD is a set of requirements. Each requirement is introduced individually into the Requirements Base and linked with related objects in the Knowledge Base (Domain Representation). The present LESD architecture [Cas-94a] is structured in two parts: the first one consists of requirements syntactic, semantic and domain analysis; resulting in the requirements conceptual representation to be included in the Requirements Base. The second one consists of reasoning algorithms about the conceptual representation of the requirements.

The work done till now in LESD has been the development of tools for syntactical and semantic analysis, the design of an appropriate knowledge representation system (using a frame-based formalism) and the study of some quality factors about the specification. We have selected five quality factors that we think are the most important ones: traceability, modifiability, completeness, consistency and verifiability. The study and the algorithms to control the first one are fully explained in [Tou-92]; the other four factors are the objective of the Spanish CICYT project TIC93-420. Modifiability has started being studied from a metrics perspective [Cas-94b], and this paper reports our symbolic approach to the control of completeness.

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. Most times this is done unconsciously by both specification-writers and specification-readers. So, these gaps in the specification can result in problematic design and code changes. As stated in [Pre-87], it is really important to detect errors as soon as possible. In fact, this was the main reason that motivated the LESD project [Bor-91].

Our intention in this paper is to present an approach to avoid this problem by checking automatically whether a specification is complete or not, and in the second case, reporting what kind of information is missing.

2. PROBLEM DEFINITION

To get inside the problem we will start by defining it. According to [IEEE-84] standard, we can say a specification is said to be 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 says too many things and is too concrete to reason in an abstract manner about it. We propose this more compact and generic one: a specification is complete if all the information needed to know the exact behaviour of the specified system is stated. For this definition to be considered accurate, two questions must be answered:

- What information do we need in order to know the exact behaviour of something?
- What do we understand as "exact" behaviour?

The first one brings up the problem of deciding which pieces of information are relevant and which ones are not. Following the [IEEE-84] standard we could answer that relevant information consists of functional requirements, performance requirements, etc. But we could use a different standard with other kind of relevant information.
The answer to the second question depends on the kind of specification we are dealing with. If it is a preliminary specification, we cannot expect every detail of the system to be present in it; so, we must only look for information conforming the system skeleton. The more detailed is considered to be the specification, the more information we must check to be present in it. It is like having several sieves with different hole sizes each one.

More precisely, taking into account the [ESA-91] and [IEEE-84] standards, and the paper [Cor-89], 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 two mentioned standards do not specify it.

3. No information is left unstated or to be determined. The use of TBD’s (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 [Tue-93].

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.

3. 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 it 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 [Reu-91], 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. So, 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.

Another way we can try to catch external incompletenesses is through traceability, either backward or forward [Tou-92]. In respect to backward traceability, the requirements in a specification are normally broken in subrequirements. These subrequirements give us more

These incompletenesses will not be internal to the specification as a whole but internal to the specification plus the domain knowledge.
concrete information about the parent requirement. Or maybe they give us information about a specific issue (for example security, performance...). In any case, backward traceability allows us to check that every requirement has been taken into account in a more detailed specification phase [Tue-93].

Because our system must deal with preliminary specifications, we must be aware of reporting incompletenesses due to the absence of requirements that would be too specific to fit in the treated document¹. A good solution to this problem would be to fix a maximum specificity level by looking to the specificity of the requirements present in the document. This level of specificity could be fixed manually by the user previous to the specification analysis. We could also try to fix the document domain in the same way (security, data treatment, performance...).

By other hand, forward traceability allows us to catch situations in which specific requirements do not have an antecedent in a more general specification level [Tou-92].

Besides traceability and adding metaknowledge to our Knowledge Base, there is still another kind of checks we can explore to detect incompletenesses. We can build a map of used entities. Everything referenced and defined in the specification must be useful in some way to define the problem treated and/or to get the goals of the specified system. See [Yue-87] for the definition of goal and a deeper analysis of what Yue calls sufficient completeness and pertinence. If we find that an entity is isolated from the final system goals or the problem definition, we can state that:

1. This entity is not useful for describing system purposes, and then must be suppressed from the specification⁴ (the entity is not pertinent to the problem specification). Or,

2. There are some requirements missing from the specification that should link this entity to the system goals/problem definition (the specification is insufficient).

Of course, we cannot assure that every incompleteness will be caught. In fact, most won’t be. So, as stated in [Reu-91], the end-user has to be the final arbiter of completeness.

4. INTERNAL COMPLETENESS

The internal completeness ensures that all the information present in the document is completely defined. Cordes and Carver propose in [Cor-89] 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 [Reu-91] we see also 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 to check also some properties resulting from the reasoning about the conjunction of the domain knowledge and the overall knowledge we have about the specification²; i.e. the Requirements Base. In this way, we catch incompletenesses internal to the specification and internal the the conjunction of the specification plus the domain knowledge.

Our initial idea is to establish this set of properties that must be checked through the use of metarequirements: requirements introduced by the domain-knowledge itself. These metarequirements specify the quality properties we are talking above 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.

¹ We assume that requirements corresponding to different specification levels are possibly in different documents.

² With these checks we also verify, in the measure we can, minimality property. That is, everything present in the specification is useful in some way to reach problem goals or define them.

³ These properties are intended to express specification quality factors (at this moment, only completeness).
No distinction will be made between requirements and metarequirements. This way, the
specification itself 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'll know that every audio emergency signal defined may have some conditions defined and
each one of these conditions must have a specific tone.

The difference between incompletenesses internal only to the specification and internal to the
specification plus the Knowledge base is related to the reasoning done in each case. In the first
one, metarequirements tell us directly about the incompleteness. For example, the
metarequirement: "Every system needs an I/O device" tells us to check every system definition
for the presence of an I/O device. No additional reasoning is needed. In the second case, a
previous reasoning phase using the Knowledge Base is needed. In the metarequirement "Every
system needs an I/O device in order to be controlled" we need a previous reasoning phase to
know whether a system is controlled or not.

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 (we are not sure about
them, but they will probably be). The modality of a requirement, used in [Tou-92] 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 sure incompleteness or warn
about a possible one.

4.1. COMPUTING INTERNAL COMPLETENESS

A first thought to compute internal completeness can be to check every completeness
property expressed through a metarequirement.

We have requirements referent to the specification (that we call metarequirements) and
requirements referent to 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) when
the library management system must perform any operation (given by a system requirement)
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 could 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 operative way.

The first one 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

*Although the specific of the tone relates to consistency, its presence is a completeness issue.
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 will know that books are repository
elements. The next step is deciding whether the specified system needs this property being
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, book definition is checked for it.

The second way is more artificial, and is based in 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 a special case of
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 some times, 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 done 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 into the combination of three different situations that
require only specific completeness checks for each one. Taking profit from this fact, we can
manage to perform all checks only once. So, the intersection between the checks applied in any
of these situations and all the checks done previously is the empty set. Let's see the situations,
the corresponding checks and why they are performed only once.

A property will be checked when:

1. The new requirement is in fact a metarequirement, and the property specified by it is
   needed by any requirement activity. The property check couldn't have been done before
   because we hadn't the property before this new requirement has been introduced.

2. A new requirement introduces an activity that "activates" some property we already had
   in the Knowledge Base\(^1\). The property was not checked before because although we had
   the property in the Requirements Base previous to the new requirement treatment, it was
   not active\(^2\).

3. A new object is created in the Knowledge Base. As this new object didn't exist before,
   no check concerning it has been done. Then, every active and related property is checked
   for that object. From this moment on, only new checks (introduced by points 1 and 2)
   will be performed for this object.

Combining these three different cases, we can construct a quite simple algorithm which
skeleton is as follows:

- Retrieve the main activity associated to the requirement (it can be either a system
  requirement or a metarequirement).

\(^1\) We say a property is activated when a new requirement activity needs it but it was not needed by any
requirement activity that was previously in the Requirements Base.

\(^2\) We could argue that properties may be deactivated by new knowledge; and so, a property could be
activated, deactivated and activated again (and then checked twice). But this behaviour seems impossible for
completeness properties if we remember what we've said earlier: new knowledge can convert orthogonal
incompletenesses to non orthogonal ones (so that properties will be activated to detect them); but never can
convert non-orthogonal incompletenesses to orthogonal ones. So, we can deduce that a property will never be
deactivated.
- 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 for performing another activity, the property is checked.
  - Operational: check if the activity can be done (this implies checking properties that condition the activity).
  - For any new entity referenced (and so, created), perform all operational and static completeness checks.

This algorithm allows us checking the completeness while introducing the requirements one by one. This can be very useful for an interactive tool.

4.2. PROPERTIES

By the moment, and referent to internal completeness, the algorithm only treats two different properties: to have an attribute and to be a concept. Both are represented by their homonymous activities.

The checks performed are as follow:

- to have an attribute: a specified entity must have defined an attribute equal or more specific than a mentioned attribute.

- to be a concept: a specified entity must be an instance of the mentioned concept or a more specific one. This check may involve both completeness and consistence. We could think we have incompleteness when the agent is an instance of a superclass of the concept. By other hand, we have inconsistence when the relation between the mentioned concept and the concept that has the referred entity as instance is not a sub-superclass relation.

If we decide, in a future, to treat external completeness relating traceability and unused entities/requirements using metarequirements in the same way we have described above, new completeness properties should be created in order to provide a more powerful reasoning system.

5. ENVIRONMENT

Our system is based upon a Knowledge Base. Initially we only have the domain knowledge and, when we read the requirements in the specification, we incorporate them to the Knowledge Base. In fact, we call Knowledge Base to the domain Knowledge Base we have initially and Requirements Base to the knowledge generated by the requirements conceptualisation. At this moment the part of the system that is working, is implemented under a frame environment: Knowledge Craft. See [Tou-92] for a complete description of the Knowledge Base.

In addition to our frame system, it would be convenient to have some more features for our environment. In [Ric-92] we can see an example of an environment, called CAKE, that is more than a frame system, incorporating automatic retractability, truth maintenance, boolean constraint propagation and other interesting capabilities.

We can separate the features among:

a) Those that are needed previous to the frames creation; that is, in the step between the semantic interpretation and the frame creation and incorporation to the Requirements Base. We can distinguish:

- knowledge unification: The system must know when a new frame refers to an entity already present in the Knowledge Base. If the new frame represents the same entity that another frame in the Knowledge Base, the information from both frames must be unified. Knowledge unification is crucial to check for specification completeness. Typically, an entity will be defined and referred in several specification requirements.
If unification were not used, we would obtain a resulting Requirements Base with several frames defining partially the same entity. The result of these partial definitions would be to report more incompletenesses than those contained in the specification. Besides important for completeness, this point is also very important to detect inconsistencies.

- hierarchy refinement: We cannot expect to have the complete domain taxonomy defined from the first moment. So, when necessary, new concepts in the hierarchy must be automatically created. When a new concept is found in the specification, it must be inserted in its correspondent location in the domain taxonomy. This insertion means linking the new concept to its parent (or parents), and possibly, redefining some parent children to be children of this new concept. This is specially important for the most specific concepts in the hierarchy; that is, those concepts that are firm candidates to become the parent of some instances. As all the reasoning to check completeness will be done through instances, we must specify perfectly the concept to which they belong.

b) Those that apply once the frames have been created. We can distinguish the following ones:

- automatic classification: Sometimes a concept comes classified through a hierarchy, but we are interested in classifying it in another orthogonal hierarchy. For the library example, we may want to classify book as a repository-element in order to detect that checking out books is more specific than checking out repository elements and thus, to detect the need for a unique identifier in books.

- order independence: The result must always be the same for the same set of requirements and for any order in which they are presented. [Tou-92] provides a classification algorithm that resolves this problem attending to the Requirements Base hierarchy automatic construction.

- retractability: It is desirable, in an interactive system, to allow retractability about requirements and definitions. In our frame system we could use some kind of metaknowledge (such as that introduced in [Tou-92] about the responsible of the frame creation) in order to make it possible. This and the preceding property are given a lot of importance in [Reu-91]; they seem almost necessary for an interactive tool.

As for the completeness treatment we treat directly with frames, the features that we must take into account are this second group. Of course, the first group is also necessary, but features conforming it are transparent to completeness treatment.

6. OTHER KINDS OF COMPLETENESS

Everything that has been said until this moment refers to structural completeness. This is also the kind of completeness treated by most systems that appear in the literature. But there are other kinds of completeness we could consider.

The most important appears to be logical completeness: the specification must contain the response of the system to any possible input in any possible situation [IEEE-84]. As a result of this definition, it is important to specify the behaviour of the system for invalid inputs. However, this kind of completeness is more important at a detailed specification level, when program states (possible situations referred in the definition) become evident and clearly differentiated. As our system is supposed to deal mainly with preliminary specifications, we will rarely find cases in which logical completeness treatment will be clearly applicable.

In some papers about requirements verification, we find documental completeness; that is, completeness at the level of document structure, not at the level of its contents. Mainly this involves references inside the document: figures, tables, index, etc.

We can also talk about methodology completeness [Woo-90]. Different to the other kinds of completeness we have seen until this point, this one refers not only to the specification, but to the global process of understanding the specification; that is, the communication act between those who have written the specification (specification-writers) and those who must understand
it (specification-readers). So, we can see the methodology completeness as a special case of conceptual completeness, that allows to transmit information from a sender to a receptor without loss of information [Tue-93].

Methodology completeness is a prerequisite for specification completeness. If the methodology is not complete, we will find problems for which every "plausible" specification will be incomplete. In fact, these problems will be unspecifiable using this methodology. So, methodology completeness is very important for us.

Since methodology will be the same for any treated specification, we must assume its completeness when it is chosen and assume it for the rest of methodology life.

We can decompose the methodology completeness into:

a) The expressive completeness of the knowledge representation system (KRS) used by the sender.

b) The power of the knowledge acquisition system (KAS) used by the receptor.

c) The expressive completeness of the KRS used internally by the receptor.

Both KRS must be able to represent everything a set of requirements could contain; while the receptor KAS must be able to read from the sender KRS everything everything a set of requirements could contain and "translate" it to the receptor KRS.

In the methodology assumed by LESD, the KRS used by the sender is always natural language. Every necessary element to specify a computer system is present in natural language [Woo-90]. So, natural language is a complete KRS for sender purposes. See the same paper for a discussion about the completeness of standard analysis methodologies and the comparison with a natural language based methodology.

The receptor is a set of one or more specification-readers (programmers, designers, testers...). Then, the completeness of paragraphs b and c depends entirely upon them, and will be related with their intelligence and experience they have about the subject. We can help to reach conceptual completeness by modifying and restricting the KRS used by the sender. This way, we can use a restricted natural language with the intention to be more concise, avoid ambiguities, etc. These restrictions must be strong enough to allow easy and total understanding by specification-readers and weak enough to still allow this restricted natural language being a complete KRS for specification-writers purposes.

In fact, it could be noticed that ensuring the completeness of the methodology as we have described here is the purpose of the overall LESD project.

7. CONCLUSIONS

We have presented here an approach to the control of completeness that deals with internal completeness and also propounds three different mechanisms to detect external incompleteness:

1. Adding metarequirements and simple completeness properties to the Knowledge Base to find out incomplete definitions. Incompletenesses detected this way are said to be internal to the specification plus the Knowledge Base.

2. Using traceability to detect gaps in the requirements map. These checks allow us to detect the absence of requirements deduced from other requirements in the specification.

3. Checking the use done in the specification of every entity defined in it. These checks may tell us that there are missing requirements related to an entity; but give no idea about what these requirements are about.

A prototype incorporating metaknowledge and completeness properties has been implemented to detect incomplete definitions. Points 2 and 3 must be further explored; and implemented, if possible, through the use of metarequirements, with the addition of new completeness properties and, if necessary, the strengthening of metarequirements mechanism. Metaknowledge and completeness properties are also used to compute internal incompletenesses.
The prototype has been tested with several simple specifications in order to validate the model. More complex specification examples will be tested in parallel with the implementation of points b and c.

8. BIBLIOGRAPHY


LSI–94–2–R "Degrees and reducibilities of easy tally sets", Montserrat Hermo.


LSI–94–4–R "Una modelización de la incompletitud en los programas" (written in Spanish), Javier Pérez Campo.


LSI–94–8–R "A transformation scheme for double recursion", José L. Balcázar.


LSI–94 13–R “Bases de dades bitemporals” (written in Catalan), Carme Martín and Jaume Sistac.


LSI-94-18-R “Medición del factor modificabilidad en el proyecto LESD” (written in Spanish), Núria Castell and Olga Slávko.

LSI-94-19-R “Algorismes paral·lels SIMD d’extracció de models de fronteres a partir d’arbres octals no exactes” (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI-94-20-R “Una paral·leització SIMD de la conversió d’objectes codificats segons el model de fronteres al model d’octrees clàssics” (written in Catalan), Jaume Solé and Robert Juan-Arinyo.

LSI-94-21-R “Clausal proof nets and discontinuity”, Glyn Morrill.


LSI-94-24-R “On infinite sequences (almost) as easy as π”, José L. Balcazar, Ricard Gavaldà, and Montserrat Hermo.

LSI-94-25-R “Updating knowledge bases while maintaining their consistency”, Ernest Teniente and Antoni Olivé.

LSI-94-26-R “Structural facilitation and structural inhibition”, Glyn Morrill.


LSI-94-29-R “Towards a VRQS representation”, Mariona Taulé Delor.


LSI-94-32-R “LDB/LKB integration”, German Rigau, Horacio Rodríguez, and Jordi Turmo.

LSI-94-33-R “Learning more appropriate selectional restrictions”, Francesc Ribas.

LSI 94 34 R “Oracles and queries that are sufficient for exact learning”, Nader H. Bshouty, Richard Cleve, Ricard Gavaldà, Sampath Kannan, and Christos Tamon.
LSI-94-35-R “Acquisition of lexical translation relations from MRDs”, Ann Copestake, Ted Briscoe, Piek Vossen, Alicia Ageno, Irene Castellón, Francesc Ribas, German Rigau, Horacio Rodríguez, and Anna Samiotou.


LSI-94-38-R “On the integration of CAD and CAE”, C.M. Hoffmann and R. Juan-Arinyo.


LSI-94-40-R “Between logic and probability”, Ton Sales.

LSI 94 41 R “A sequential and parallel implementation of skip lists”, Xavier Messeguer.


LSI-94-44-R “A note on genericity and bi-immunity”, José L. Balcázar and Elvira Mayordomo.


LSI 94 46-R “Prototipatge semantic d’un model conceptual deductiu” (written in Catalan), C. Farré and M.R. Sancho.

LSI-94-47-R “Generació i simplificació automàtica del Model d’Esdevenits Interns corresponent a un model conceptual deductiu” (written in Catalan), C. Farré and M.R. Sancho.

LSI-94-48-R “B-Skip trees, a data structure between skip lists and B-trees”, Joaquim Gabarró and Xavier Messeguer.

LSI 94 49 R “A posteriori knowledge: from ambiguous knowledge or undefined information to knowledge”, Matías Alvarado.


Copies of reports can be ordered from:

Nuria Sánchez  
Departament de Llenguatges i Sistemes Informàtics  
Universitat Politècnica de Catalunya  
Pau Gargallo, 5  
08028 Barcelona, Spain  
secrelsi@lsi.upc.es