The Completeness Problem in LESD

Antoni Tuells
Núria Castell

Report LSI-93-51-R
THE COMPLETENESS PROBLEM IN LESD

Antoni Tuells Jansson
Núria Castell Ariño

Dept. LSI, Universitat Politècnica de Catalunya
Pau Gargallo, 5
08071 Barcelona
Spain

e-mail: atuells@lsi.upc.es
e-mail: castell@lsi.upc.es

Abstract

This paper presents an explanation, classification and some solutions to the completeness problem within the LESD project. The aim of this project is to develop a set of software tools able to help users to write good quality aerospace software specifications. Completeness is one of the quality standards (together with traceability, consistency, verifiability, and modifiability) we are dealing with in LESD.

Resum

Presentem aquí una explicació general del problema de la completeness, junt amb diferents definicions d’aquest concepte. A continuació tractem el problema de la completeness en el marc del projecte LESD, proposant algunes solucions. L’objectiu d’aquest projecte és desenvolupar un conjunt d’eines per a facilitar la redacció d’especificacions de software en el domini aeroespacial. La completeness és un dels cinc factors de qualitat considerats a LESD (junt amb traçabilitat, consistència, verificabilitat i modificabilitat).
0. Introduction

In this document we present an explanation, a classification and some solutions to the completeness problem within the project LESD ([19]). The aim of this project is to develop a set of software tools able to help users to write spatial software specifications. These specifications are intended to fit some quality standards (completeness, consistency and verifiability) [19].

In his classic paper, "No Silver Bullet" [4], Brooks identifies four essential difficulties in building large software systems as follows:

* **Complexity**: The complexity of the development process, the application domain, etc.
* **Conformity**: Software cannot have a regular formal structure (like Maths or Physics), as there are a necessity of having Natural Language documents in the early stages of software development. In other words, we cannot avoid the use of Natural Language in building large Software Systems.
* **Changeability**: In any system, the software component is usually the easiest to modify.
* **Invisibility**: The structure of software is hidden. The only external evidence we have of software is its behaviour when executing.

Regrettably, the more complex and large a system is, the more difficult it is to measure its quality, due to the fact that the difficulties we have mentioned above grow up together with the size of the system. This is essentially true with respect to measuring the completeness degree of Software documents. The first problem is to establish a formal definition of Completeness, as it seems that many authors misunderstand its meaning. The meaning of "completeness" is usually bad defined and not clear, especially if it is referred to Software Engineering, probably because it is used in many and different fields of Science. We use the word "completeness" in Logic, Linguistics, Software development and Knowledge Representation, and we will show that it is used with different meanings, so it is well worth to make some kind of classification among them.

1. Some trends in Software Engineering

We present in this section some trends currently appearing in Software Engineering [10,21], paying special attention to its relationship with Natural Language, Software Requirements and Completeness. More specifically, we explain the importance of Natural Language in Software Requirements, the relation between Natural Language and Formal Languages and, finally, we introduce the Completeness problem.

Computer-aided software engineering (CASE) is undergoing tremendous commercial growth. Unfortunately, the current generation of CASE tools is limited by shallow representations and shallow reasoning methods. These tools will be probably replaced by tools including more sophisticated reasoning methods and deeper representations [3,5,6,11]. The technology enabling those tools will come from AI and other areas of computer science (this technology is usually known as Knowledge Based Software Engineering, i.e KBSE).
Currently, research in intelligent assistance for requirement and specification engineering is less mature but already shows considerable promise. Software will become adaptive and self-configuring, enabling end users to specify, modify, and maintain their own software within restricted contexts. In essence, software engineers will deliver the knowledge for generating software rather than the software itself. Capturing structural and syntactic information from a software design expressed in Natural Language into a complete semantic representation remains an open question.

As hardware performance increased, the scope of software projects soon exceed the capabilities of small teams of programmers. As coordination and communication become dominant management concerns, strict quality control in terms of a formal model in software requirements are unavoidable. In other words, we are not able to express coordination and communication among people in mathematical terms and, as a result, we have to deal with these problems using other techniques.

Natural Language and Formal languages play important and different roles within the process of software development. Natural language is often used in the early stages of the process, where it serves as the common tool for problem expression [1]. It is not possible to avoid its use because software requirements involve many people having different backgrounds, and Natural Language is often the best option for communication purposes. In addition, the process of software development needs all mental resources of the engineer, and using Natural Language seems to be the best option for. In fact, software requirements in complex systems are usually written in Natural Language [20]. These documents might be used in the next stages of the software life cycle if necessary (when executing the software or for maintenance purposes). As a result, software engineers usually work with and have to take into account the original documents. Thus, it is desirable to make some effort for writing good quality software requirements, as these will be always used. Standards have been defined by IEEE, ESA or NASA to guide specification writing. These standards specify two main kinds of constraints: linguistic and software engineering constraints. Linguistic constraints are intended to limit irregularities due to Natural Language (polysemy, paraphrase, ambiguities and vagueness) and take the form of informal recommendations: a requirement should be a simple sentence, avoid vagueness and ambiguity through the use of a restricted vocabulary. Software constraints are concerned with all software aspects of a system, including mandatory practices, guidelines and milestones.

Formal languages imply a previous training before using them and serve for other purposes than Natural Language. Usually, people use Natural Language for explaining what the system has to do (i.e. the system functionality), and Formal Languages are used for describing the external software behaviour in terms of a formal system [2,7,21]. A formal system consists of both a formal language (i.e., an alphabet and syntax) and a deductive apparatus for manipulating strings defined in the formal language. Acceptable strings in the language are referred to as well-formed formulas (wff), and meaning (or a semantics) is given to the system by assigning an interpretation to each wff in the language. It is important to point out that the deductive apparatus makes no reference to a particular interpretation, and is based purely on syntactic manipulation. Within a formal system, a proof is simply a finite sequence of wffs that are either axioms or direct consequences of an inference
rule. An important problem arises, namely, assigning interpretations to wffs; we prefer those that are both consistent and complete (checking completeness usually means trying to demonstrate some theorem within a mathematical model). However, we may use incomplete specifications as a mechanism to postpone certain decisions about software behaviour. [7]

Obviously, there is a need for bridging the gap between Natural and Formal Languages. One interesting approach is the one due to [15], that consists in translating sentences from a pseudo-natural-language into a formal language. Unfortunately, the authors restrict the natural language in a way that they make it unsuitable for real problems. Another approach is interactivity and conceptual models of the application domain. Why interactivity? Due to the absence of automatic methods for bringing informal specification into formal ones, and as requirements are assumed to be incomplete until late in the development cycle, because in the early stages of a project the total design is not completely understood, a great many of authors have suggested the use of interaction (or prototyping) between the computer and the user for this purpose. To put it another way, the software engineer has to be the arbiter of the situation and has to decide whether software specifications are complete. As an aid for checking completeness, some authors have proposed Natural Language as a guide to the development of conceptual models, as it seems that up to now these are strongly influenced by the methodology.

What about Completeness? A common definition of this term in Software Engineering is the following: Completeness is the ability of capture the system functionality from the users. It is difficult to assure as there are quite a large number of problems involving the activity of extracting information from the customers or users. Let’s see some of them:

* user’s incomplete understanding of needs
* conflicting views of different users
* users poor understanding of computer capabilities and limitations
* analysts poor knowledge of problem domain
* it is easy to omit “obvious” information
* requirements evolve over time
* unnecessary design information may be given.

There are basically two main trends in Software Engineering dealing with completeness: namely, the first one tries to solve the problem in some way [9], and the second does not; some authors believe that it is not possible to deal with that problem (in other words, they believe that it is too difficult to assure that some specification reflects exactly the user requirements, no more, no less) and do not have any intention to solve it. Fortunately, other authors do not complain about the difficulty of the problem and feel able to try to offer some solution to it. All offered solutions have in common two features:

* These provide a partial solution.
* The role of interactivity is essential.
2. Some types of Completeness

This section is devoted to the classification and relationship of some different meanings of "completeness", and surely it will be not exhaustive. In other words, probably we have not gathered every meaning of "completeness", but we hope that our classification will bring some light to this problem.

2.1 Deductive Completeness

This is the meaning of completeness that people use when they are referring to a formal system, or the meaning Godel used in his famous Theorems. This might be a possible definition:

"A formal system is complete if for every proposition we can establish its truth value."

From a scientific point of view, the completeness of a theory is closely related to its prediction capacity. The more complete a theory is, the more events it can predict. Roughly speaking, it serves as a degree of quality of a specific theory, and it is usually used as a stimulus to improve it.

2.1.1 Some solutions to the lack of completeness

a) Changing the theory in order to increase its prediction capacity. We have to add some hypothesis to our theory or change some of them.

b) Closed World Assumption. If our system cannot infer A, then it assumes ~A. Data Bases usually use this method to answer queries. For instance, if we have a relationship among students of a University, we assume that X is not a student of this University if X does not appear within this relationship.

Obviously, this method is very useful if we have a great many of negative information. From the logic point of view, we are assuming as a ~A every undecidable element A.

2.1.2 Some comments to the solutions above

a) We cannot pretend that a theory is able to predict every event. To put it another way, no theory is complete, but a theory is complete enough if it can predict every event we want to predict.

b) As we said above, usefulness of this method is strongly related to the quantity of negative elements from the world we are treating. The more negative elements there are, the more useful it is.

2.2 Expressive Completeness

The meaning of completeness used in Programming Languages, Linguistic Formalisms or knowledge Representations systems is a meaning about its expressivity or about its lack. This idea of completeness is dependent of what we want to express; it is a matter of degree.

Suppose we have a program written in Prolog, in which we have some predicates and facts. One of them is the following: mother(pere, joana), which tries to express the relationship between two people: namely, Pere and Joana. This predicate is unable to show all meanings from a relationship between a mother and her son, but it is useful enough to know the mothers name of a specific person.
It is important to point out that expressive completeness is a matter of ability to what we want to express, to the knowledge types we try to represent and which are necessary in our domain. We distinguish between two kinds of expressive completeness [16]:

a) A formalism (for instance, a programming language) is functionally complete if it is able to describe and codify the system functionality we are modelizing. From this, as we believe in Church-Turing thesis, every computation problem can be solved using (does not matter which one) a programming language.

b) A formalism (for instance, a programming language) is notionally more complete than another one if at least, we can directly describe a problem within it, and we cannot within the second one. For instance, suppose we have two programming languages, Pascal and Pascal', where the second differs over the first only in one construction: “Repeat...until”; Pascal' does not allow the use of this instruction. Obviously, both languages have the same expressive functionality, but the first is notionally more complete than the second.

2.2.1 Some solutions to the lack of completeness

a) Increasing the formalism expressivity, both functionaly and notationally. Usually, that process means a cost: many problems turn to be computationally undecidable as we increase the expressivity power of the formalism. We can specify a great many of problems inside our formalism, but we cannot compute it.

A good example is Terminological Logics [12]: It serves as a formal apparatus in order to formalise the notion of frames as structured types, often called “concepts”. The most important operation in terminological logics is determining if one concept subsumes another.

Informally, one concept subsumes another if the first is more general than the other. In any situation, all instances of the subsumed concept must also be instances of the subsuming concept.

For example, the concept
“person with at least two children” subsumes the concept “person with at least three children who are lawyers”.

Subsumption is difficult to compute correctly in reasonably expressive terminological logics. Thus there is an unfortunate trade-off between expressive power and computational tractability in terminological logics based on standard semantics.

2.3 Structural Completeness

This is the sense of completeness we use when we speak about the lack of a structure within a more general one.

A good example is LFG [16] (a linguistic formalism) where, a semantic form represented by feature “pred” is only complete if every grammatical functions that this feature relate have specific values in the functional structure. To put it another way, feature “pred” is a structure that is complete if every substructure (i.e grammatical function) is complete.

There are more interesting examples from the AI point of view: probably, one of the most famous
is the Scripts approach [14]. Scripts try to modelize real life facts that are suppose to always follow a specific pattern. Oriented-tasks Dialogs [8] are in fact similar to Scripts; both differ in the world they are dealing with, but they follow an identical pattern: Trying to modelize worlds using fixed structures. Obviously, these approaches are suffering from the “Closed World Assumption”: To put it another way, they may be useful in a small closed world.

Let’s see a fragment of a task oriented dialogue taken from [8]:

E: Engine       A: Apprentice

E. First you have to remove the flywheel
A. How do I remove the flywheel?
E. First, loosen the two allen head setscrew holding it to the shaft, then pull it off.
...

Several issues are important to be mentioned:

a) The “small” world the dialogue is dealing with.
b) The shared Belief between the Apprentice and the Engine. It follows from a).
c) Tasks are assumed to be broken down in several subtasks.
d) Not all the subactions need to be introduced explicitly into the discourse. The Apprentice may do several that are never mentioned, and the expert may assume that these are being undertaken on the basis of other information that the apprentice obtains.

These structures have the advantage (or disadvantage) that they can make inferences about the world modelizing. For instance, in the Oriented-tasks-Dialogs approach, it is not obligatory that the user exactly follows the task-subtask hierarchy. The inferring process might infer that the user could have finished the previous tasks in the hierarchy. For this purpose we need a user-behaviour model that serves to implement the shared beliefs between the system and the user.

2.3.1 Some solutions to the lack of completeness

a) Filling up the structure or improving the inference processes properly.
b) Making the world we are dealing with “smaller”, in order to have a structure as less complex as possible.

2.4 Conceptual Completeness

This is the sense of completeness we use when describing communication acts between different people. According to this point of view, a communication act is complete if the receiver of the message can understand it.

From this, software specifications written in Natural Language are complete if software engineers understand or are able to infer the system functionality (what the system has to do, but not how the system should do it).
As people that writes specifications are not the same than the ones that have to read it, it is very important and useful to write software specifications as complete (conceptually complete) as possible. As a result, it is worthwhile to make an effort for writing software requirements; this is the reason for the presence of so many "how to write good quality software specifications" standards [17].

As we can deduce, the conceptual completeness problem is a linguistic problem (how the message is send) and a cognitive one as well (the cultural background of the scientists play an important role, as well the shared belief that is supposed to exist among people working in the same project). As we can see from the example of the oriented task dialogue, the shared knowledge about actions and objects in the domain of discourse is especially important when the linguistic markers are insufficient.

There are other problems we have not mentioned yet: Studies about cognitive processes underlying systems analysis have described categories of mental behaviour for expert analysts [18]. These studies do not propose any models of the systems-analytic reasoning process. Instead, they discussed on correlations between frequencies of mental behaviours and expertise exhibited by experienced analysts. More effective gathering of domain information, better formation of conceptual models of the problem domain and more critical testing of hypotheses have been suggested as qualities which differentiate expert from novice analysts. Psychological studies of program designers have demonstrated that novices have fewer preformed mental schemes which can be retrieved from memory and that novices tend to focus on the surface aspects of the problem rather than the semantic level of the problem itself.

To summarise, it is difficult to measure how complete software requirements are due to the factors that are involved: linguistic, shared knowledge and expertise.

2.4.1 Some solutions to the lack of completeness

a) Improving the linguistic quality of software requirements.

b) The software engineer has to acquire information and build a model of the domain before design can proceed.

3. Some Approaches to the completeness problem within LESD.

The project LESD deals with huge documents of software specifications written in English; as a result, deciding whether these specifications are complete turns to be a complex matter. There are three major approaches for solving the completeness problem inside LESD:

* The traceability approach
* The scripts approach
* The task-subtasks approach

1) From the traceability point of view [19], and according to standard ESA-PSSO5, completeness is assured by backward traceability. However, it seems that backward traceability is necessary but
not sufficient to assure that a set of software requirements is complete.

Certainly, backward traceability allow us to be sure that in a specific phase of the design process, every requirement has been taken into account. Regrettably, original and huge documents are usually separated into many others, each of them related to a specific topic (data manipulation, security, communications,... see figure). Backward traceability cannot assure that all needed requirements from the original document have been taken into account in each of the documents devoted to a more specific topic. Furthermore, the number and kind of requirements needed varies respect with the expertise of the software engineer (section 2.4).

A partial solution inside the traceability approach might be to trace all the requirements according to a specific feature (security, ...).

Forward traceability allow us to treat another side of the completeness problem, as if a requirement does not have an antecedent in a higher level, the document is probably not complete.

2) The Scripts approach does not work in LESD as the system does not have a complete knowledge of the domain. The initial Knowledge Base contains a partial description of the domain and new objects and actions are added as new requirements are analyzed; in other words, the Knowledge Base is built in an incremental way. As a result, the knowledge about actions and objects appearing in the spatial domain is only partial [19].

Furthermore, it is not clear how to define a global script fitting all possible requirements within a spatial domain.
3) From the task-subtasks point of view, it seems interesting to study the possibility of building an action-subaction hierarchy, which could help the user for assuring a minimal "complete degree" of the software requirements. This hierarchy should be application-independent and should be built up out of a corpus of requirements belonging to different applications.

The building of the hierarchy is not straightforward, as it is important to mention that activities (i.e verbs) select their arguments (i.e objects), and it is not clear to build this structure in an activities arguments-independent way. Probably, we will have to obtain the task-subtasks hierarchy in a semi-automatic way, providing we are able to build it, as it seems that there are not powerful tools for doing this job in an automatic way.

An example in our Knowledge Base is activity "monitorize", which can be decomposed in other three subactivities: "receive", "analyse" and "display".

Let's see a real requirement:
"During the launch phase, the IOI-GS shall analyse and display the status of the space vehicle"

The system analyses the requirements sequentially and builds the Knowledge Base incrementally [19]; if it detects no reference to activity "monitorize" in any previous requirement, the system outputs a message to the software engineer showing this problem.

Similarly, once the system has the semantic representation of every requirement, it suggest a completeness problem if it detects a reference to activity "monitorize" but no references to the subactivities of "monitorize".

As we use reasoning mechanism on the Knowledge Base, reliability of it is essential. The engineer has to verify and check inferences made by the system, and he has to be the arbiter of the situation, as an automatic completeness control of software specifications is not currently available, due to the fact that we are trying to capture "conceptual completeness" via "structural completeness" (i.e. the activity-subactivity hierarchy).

Currently, it remains an open question whether we can build this hierarchy or not. Let's see some examples:

Example 1:
"The IOI-GS shall provide the capability to support to an external strategic management function by producing inputs, addressing the capabilities and availability of the IOI flight and ground segments for a period at least 5 years"

From the example above we believe that we cannot suggest that activity "to provide" is decomposable into others (i.e to produce and to address) because the important facts in this requirement are the arguments the verbs take. Besides, it would be not obvious how to obtain an accurate semantic representation of this requirement for obtaining the task-subtasks hierarchy.

Example 2:
"The IOI-GS shall be able to generate, integrate and validate vehicle/crew Operations procedures"

We believe that example above shows clearly enough that activities "to generate", "to integrate" and "to validate" might be part of a more general one.

Example 3:
"The IOI-GS shall provide the capabilities to perform qualification and acceptance of each new or
modified item of the GS, and system level verification of the integrated GS including all external interfaces before its operational use”.

This example shows the complexity of analysing software requirements. Activity “to perform” takes many different arguments and, there are also references to some objects of the domain and temporal constraints. It remains unclear for us how to get a task-subtask hierarchy out of the semantic analysis of this requirement.

The examples above explained show us how difficult is to get a real task-subtask hierarchy out of a corpus of software specifications. It is still unclear and remains an open question how to assure that a set of software specifications is complete, though we are quite sure that the use of interactivity between the system and the user is unavoidable at the moment.

4. References

"Informality in Program specifications"
IEEE Transactions on Software Engineering

"Using formal methods to specify the functional properties of engineering software"
Computers and Structures

"Didon: system for specification validation"
Information and Software Technology

"No Silver Bullet:Essences and accidents of software engineering”
IEEE Computer Magazine
April 87.

"KBRA: A New Paradigm for Requirements Engineering”

"LaSSIE: A knowledge based software information system”
Comm. ACM

[7] BS Doherty, B Pauly and J Barthelemy
"Elicitation and verification of a functional specification”
[8] B. J. Grosz
"Attention, Intentions, and the Structure of Discourse"
Computational Linguistics
Volume 12, Number 3, July-September 1986

[9] M. S. Jaffe, N. G. Leveson
"Completeness, Robustness, and Safety in Real-time Software Requirements Specification"
Comm. ACM
pag 302-311, 1989

[10] M. R. Lowry
"Software Engineering in the twenty-First Century"
AI Magazine
pag 71-87, Fall 1992.

Requirements Analysis: Problems and the STARTS Approach

[12] Peter F. Patel-Schneider
"A Four-Valued Semantics for Terminological Logics"
Artificial Intelligence
Vol 38, pag 319-350, 1989

"The Requirements Apprentice: Automated Assistance for Requirements Acquisition"
IEEE Transactions on Software Engineering

"Scripts, Plans, Goals and Understanding"

"A Processing System for Program Specifications in a Natural Language"
21st Annual Hawaii International Conference on System Sciences.
Vol II: Software Track

[16] Shieber, S.
"An Introduction to Unification-Based Approaches to Grammar"

[17] R. J. Stevens
"Some Considerations on Organising Requirements for Systems Management"

[18] Sutcliffe, Maiden
"Analysing the novice analyst: cognitive models in software engineering"
International Man-machine Studies
Vol 36, pag 719-740, 199212
[19] Y. Toussaint
"Méthodes Informatiques et Linguistiques pour l'aide a la Spécification de Logiciel"

"Applying Linguistic Engineering to Software Engineering"

"Extending Analysis Paradigms"