Knowledge-Based Techniques for Software Requirements Validation

Jordi Alvarez, Núria Castell
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Mòdul C5, Campus Nord, 08034 Barcelona
{jalvarez, castell}@lsi.upc.es

Abstract

This document explains a knowledge based model intended to help in Software Requirements Validation. We first construct automatically a specification model from a set of correct specifications using machine learning techniques; and then this model is used to validate new specifications. The machine learning techniques that are used, combine ideas from terminological reasoning, analogical reasoning and probabilistic representation.

Contents

Chapter 1

Introduction

Requirements Engineering is one of the most important phases in software development. Most errors detected in coding or maintenance come from bad specifications [6]. Moreover, the cost of correcting a specification error increases enormously when it is detected while in coding or maintenance [30]. This was one of the main motivations for the LESD project [9, 12]. The aim of it was to control the quality of preliminary specifications written in natural language. LESD selected five important factors to validate specification quality: *traceability*, *completeness*, *consistency*, *verifiability* and *modifiability*. This project is continued by the CICYT TIC93-420 Spanish project. The work explained in this paper is part of it.

Because we are dealing with preliminary specifications and they are written in natural language, giving a computational definition about the quality of specifications becomes very difficult. We can try to define the quality of specifications in function of those previously stated quality factors. Those quality factors can be further decomposed into several criteria (see [34] for a study about quality factor decomposition based on metrics); but it is still very difficult to complete the theoretical study that leads us towards a computational definition of the quality of specifications.

By one hand, as we are dealing with preliminary specifications we will not have complete information about the external behaviour of the system but only an outline of it. Information will be more vague than that from detailed specifications.

By other hand, although we are using a restricted natural language (with several style guidelines in order to improve readibility and reduce ambiguity [1]), it is impossible to remove totally informality inherent to it. It is true that we do not work directly on natural language but on a preliminary conceptual representation obtained from specification texts. But not all the difficulties of informality are overcome by careful knowledge conceptualization [23]. So, we cannot trust the conceptual representation being free from those natural language undesired characteristics.

This contrasts formal specification languages, designed specially to avoid those (and other) inconviniencies. But, as formal languages focus on machine specification (easier to formalize), we focus on the description of what [23] calls the *requirement* and *domain properties*.

The first approach we took was to extend a domain knowledge base with some kind of meta-knowledge that allowed to state some conditions over domain descriptions [3, 2, 5]. We used a kind of *goal/behaviour-oriented* view of requirements as the theoretic base of our approach (see chapter 2). The conditions we put over the domain descriptions in the specification were based on the goals and behaviour the system is to achieve (described in the specification itself) and on the main goal of the overall software process: to achieve the construction of the system. The relationship among goals, behaviour and descriptions was implemented through what we called *metarequirements* (see chapter 3).

The study was done focusing specially on completeness. Because of our environment and its inherent informality, the applicability of this approach was doubtful. But even in the case a definition over some quality factor could be given, it would depend on *context factors* such as the methodology used, the standard followed, the working domain, etc. To give a hand-written definition for every context factors combination would be too expensive.

This and the informality inherent to preliminary requirements specifications (as explained before) are the reasons we propose to use machine learning (chapter 4). The idea is to have a preliminary knowledge model and then complement it with knowledge taken from a set of correct specifications (the *training set*). This enhanced model will allow us to validate other specifications more precisely than our preliminary model. The training set should be constituted by real world specifications having the same context factors that specifications to be tested have.

A *semantic networks formalism* is used to represent specifications. As most problems from software engineering can be seen as having hierarchical structure [26], a formalism allowing easy representation of this kind of structures is well suited. By other hand, semantic networks in conjunction with the three representational mechanisms stated in [7] (aggregation, classification and generalization) constitutes maybe the most appropriate formalism to represent this kind of problems [16].

We understand a requirements specification as one or more natural language texts. Its conceptual representation is the description of the overall set of objects and concepts (domain objects)[1] referenced in it. An object description is constituted by all the relationships that link that object to other objects in the specification and is extracted directly from natural language sentences.

[1] We understand the term *object* in a very general sense; referring to physical entities, activities, goals, domain concepts and so on. In fact, we mean every abstract entity represented as a semantic network node.

Chapter 2

Theoretic Study of Specification Quality Factors

The first step in the study of requirements specification validation was a theoretic study of the different quality factors we are taking into account. Unfortunately, at the present moment, the only factor we have studied from a theoretic point of view has been the completeness one. This was because the initial model we proposed (only for completeness), strongly depended on the theoretic base proposed. Later on, the model shifted to a learning system which didn't depend on the theory in respect to the detection of errors (but depends on it when deciding to which factor an error is related). So, I focussed on the development of it instead of going on with the theoretical study.

2.1 Completeness

2.1.1 Importance

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. Almost half of errors coming from requirements are due to incompletenesses [15]. People who write the specification (*specification-writers*) tend to take for granted things that seem obvious to them. When some of them are needed to build the system, persons who read the specification (*specification-readers*) fill these *information holes* with their own surmises. As the *background knowledge* that have specification-writers and specification-readers is often different, specification-readers surmises may differ from what specification-writers had taken for granted. This would cause the final system behaviour being different from the one that was initially thought.

Most times this is done unconsciously by both parts. These specification gaps are detected only when an inappropiate specification-reader surmise results in an incoherence or error in the design or the code; or when the customer realizes that the real behaviour of the system is not the expected one.

2.1.2 Definition

It is a really difficult problem to find out which information a specification lacks. To realize it, we only need to look at the definition of what a complete specification is. According to the [21] standard, we can say *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. So, we prefer to give our own definition. But before doing that, we must first remember a specification is a document to allow information interchange between different persons involved in a software engineering project. Then, if we assume that the aim of the overall 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.*

Several questions arise from the given definition. First of all, *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; that is, to contribute to the purpose of the overall software development process (build a system conforming to a set of given requirements). 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 system must be built (designed, coded, tested and maintained) by a group of software engineers. These persons must know the exact expected behavior of the system in order to construct it. So, this definition implies the specification contains all the information software engineers need in order 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.

Answering these three questions is a really difficult task. Besides, when any answer is given, it may depend on context factors such as the methodology used in the development process, the kind of specification we are dealing with, the commonsense and background knowledge supposed to be hold by software engineers, the system domain, and so on. So, it will also be very difficult to give a computational definition for the completeness of requirements specifications.

2.1.3 Hints to Necessary Information

As we see a specification as a set of object descriptions, besides talking about the completeness of a specification, we can talk about the completeness of the description of an object with respect to the specification (it makes no sense to talk about the completeness of the description of an object in an independent manner). The description of an object will be complete if all the information related to the object that is needed in order to build the system, is stated. Obviously, a specification will be complete when the description of every object referenced in it, is also complete.

Looking individually at each object, a first step to detect incompleteness is to check that every object is defined. In this sense, [33, 24] maintain a list of undefined objects. [13] checks that objects are completely integrated in the domain hierarchy (by checking taxonomic relations).

Classification

The next interesting thing we can do is to check for a minimal description of an object, depending on the kind of object it is [13]. Object-centered languages, starting with RML [19], use the classification mechanism to force the description of every instance of a class to have also a minimum description.

The introduction of *metamodels* in systems such as TELOS [29] and KAOS [14] allows to apply this check for domain concepts (classifying them as instances of metaconcepts)[1].

Unfortunately, it is nearly impossible to have every relevant property in the domain or metalevel taxonomy. So, knowing just the kind of the objects is not enough to check the completeness

[1]We can even see a kind of simple and fixed metamodel in the activity-entity RML model [20]

of a specification. This is the main reason that makes all systems we were talking in the preceding paragraph to include first order predicate logic extensions to express restrictions over the model. This allows them to perform more complicated checks.

From Goals to Structure

Providing the convenient connection between logics and the existence or non-existence of information in our knowledge base, we could express almost everything. But, what do we need to express? To answer this question, we distinguish among three different kinds of information that requirements engineering must deal with: *intentional*, *behavioral* and *structural information*. The behavioral-structural duality can be found in the activity-entity RML model and also in the JSD methodology [22]. TELOS and KAOS, besides distinguishing activities (behaviour) and entities (structure), introduce goals (intentional information) into their metamodels.

Intentional information refers to the goals that different agents appearing in the specification have. Goals may imply or be restricted by other goals [28]. This means there exists an implication inside intentional level itself. But there are other implications. To achieve goals, agents must exhibit a certain behavior. So, there exists also an implication between intentional and behavioral levels. [28] expresses this implication in a decomposition of goals into methods; and KAOS system expresses it in the *goal operationalization* [36].

The same that for intentional level, we also have an implication inside behavioral level itself. A piece of behavior may need another piece of behavior to be feasible. An easy example is an activity decomposing into sub-activities.

Then, we go further and establish an implication between behavioral information and structure. It is reasonable to believe that for a certain behaviour to happen, the objects taking part in it should have a certain structure. For example, a subsystem cannot receive information if it has not an input device. Another problem is which part of this overall information should be expressed and which part can be overlooked (question 3).

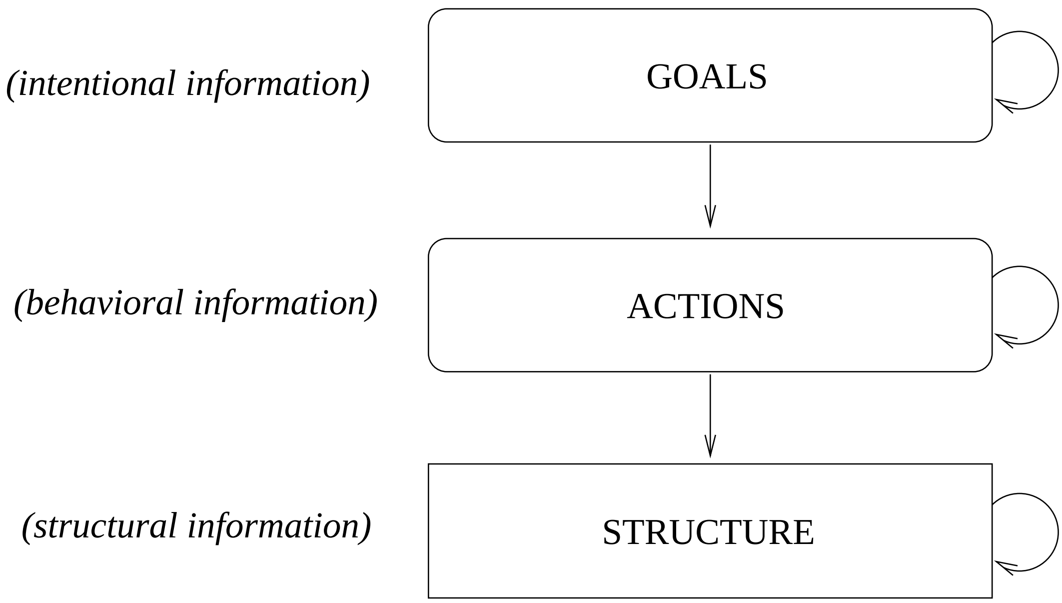

Figure 2.1: Implications among different levels of information.

Hints Combination

This set of hints conform a very generic model to check the completeness of specifications. They don't tell us which pieces of information we do need or which ones we do not need. Instead, what the hints do is to give us a domain dependant way to find necessary information. They conform the base that in conjunction with some domain dependant knowledge will allow us to check the completeness of specifications.

Chapter 3

A Metaknowledge Based Validation Approach

When software engineers read a specification, their knowledge about the domain, software, computers and so on, makes them realize some specification errors (inconsistencies, incompletenesses...). The first approach we took to validate requirements specifications was to add more knowledge to the knowledge base to make possible the automatic detection of as many errors as possible. Referent to completeness (the only factor that has been treated in this approach), the added knowledge established some kind of implications among kinds of goals, behaviour and structure corresponding to a concrete domain or to more generic concepts, relevant to several domains.

3.1 Metarequirements

This knowledge (metaknowledge, in fact) is constituted by a set of what we called *metarequirements*: requirements over the knowledge we have about the system; that is, about the specification itself[1]. These metarequirements specify information needed in specification in the same way that system requirements specify the system to be built.

Metarequirements can be seen as production rules. In the sense that, for example, in the goal-behaviour implication, when some kind of goal is instantiated, some kind of behaviour needs to be instantiated also. An the same holds for the other kinds of implication we can see in figure 2.1.

The essence of the overall idea is similar to that of metamodelling [29, 14]; but it lacks the object-centered approach that is present in metamodelling since what can be considered its beginning [8]. The difference is that instead of having a set of metaconcepts which instances must hold some conditions, we have a set of generic concepts that play the role of metaconcepts and a set of production rules that lead to the validation of subclasses of those generic concepts.

3.2 Main Problems

This approach requires a big effort from the knowledge engineer who builds the knowledge base extension. This effort is not cost-effective because, as the domain is very weak (preliminary requirements specifications written in natural language), most part of the extension built should be specific for a particular set of context factors. When one of this context factors changes, the metarequirements extension should also be changed.

By other hand, it is difficult to freeze metarequirements from implementation details. In order to know, using the decomposition of information in figure 2.1, whether the system external

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

behaviour stated in the specification will lead to some incompleteness, we need to deepen into behaviour. And sometimes this means going into internal behaviour (that is, implementation).

Of course, the set of metarequirements is the only piece of knowledge dealing with internal behaviour. They represent a bridge between the overall information present in the specification and goals and external behaviour. And sometimes, to cross from one side of the bridge to the other, we will need some pieces of internal behaviour.

This means introducing implementation details into the knowledge base (through metarequirements). This is dangerous because if we want a complete requirements validation system, we will probably need to build implementations for every piece of external behaviour possible in the application domain. And this is too expensive to be done, even for one domain.

Chapter 4

A Machine Learning Based Approach

To solve the problems introduced by the previous approach, we proposed to use a machine learning based approach. An explanation of a preliminary version of this approach can be found in [4].

It is difficult and very costly to build a knowledge base extension for specification validation purposes. So, the learning mechanism we propose tries to build this extension automatically from a set of valid specifications. As the learning mechanism is quite robust, any specification that has been used for a succesful project and has context factors similar to those we want, will be valid.

What the learning mechanism does is to refine a previous conceptual model[1]. This refinement is done in two directions. First, new concepts are generated. Concepts are generated heuristically from the information we have in the knowledge base. This is done using a terminological reasoning approach [10, 25]. We will have primitive and non-primitive concepts; and non-primitive concepts will be generated from other concepts and properties in the same way that concepts in KL-ONE [10] and LOOM [25].

Second, a description is associated to every concept. These descriptions are computed from concept instances; storing every relevant property and the ratio of instances that have that property. These descriptions can be seen as COBWEB's probabilistic concept representation [18]; although we here deal with generic properties while COBWEB deals with a predetermined set of attributes. As it has been said in the introduction, semantic networks is the underlying representation mechanism used.

The combination of terminological reasoning and probabilistic description representations is what will allow to validate the specification. Objects conforming the specification to be validated (remember we see a specification as a set of object descriptions) are classified into the learned conceptual model. Then, if the properties of the object do not match the concept probabilistic description ones, human intervention is required to solve this problem. Once a specification has been validated it can be used by the learning mechanism to refine the conceptual model. Figure 4.1 shows an outline of the overall learning and validation system.

4.1 Learning from Structure

Every specification is constituted by a set of object descriptions. We understand an object as any entity (physical or not) represented as a semantic network node. So, activities, constraints, actions, events and so on, are all of them objects.

Every description is an aggregation of more simple descriptions usually called attributes or slots. Here, we will consider always them as relations between two nodes. In this way, in figure

[1] This previous conceptual model can be as simple as consisting of an only concept/metaconcept. Nevertheless, the more complete is the previous conceptual model, the less we have to learn and the more directed will be the learning.

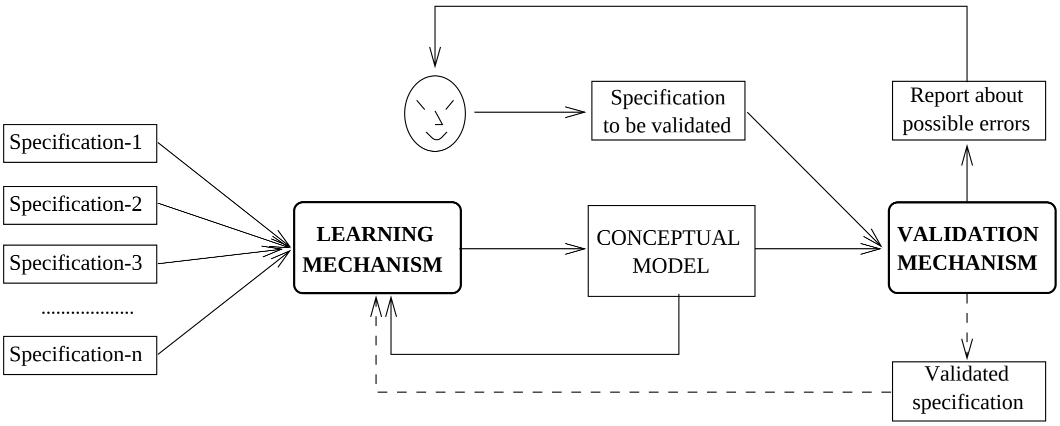

Figure 4.1: Learning and validation processes general outline.

4.2, the aggregation of CONTRIBUTES-TO, NEEDS (twice), ASSURES, PRECONDITION (twice again) and PERFORMED-BY (inverse of PERFORMS) and the corresponding nodes in the semantic network constitute the description of BORROW.

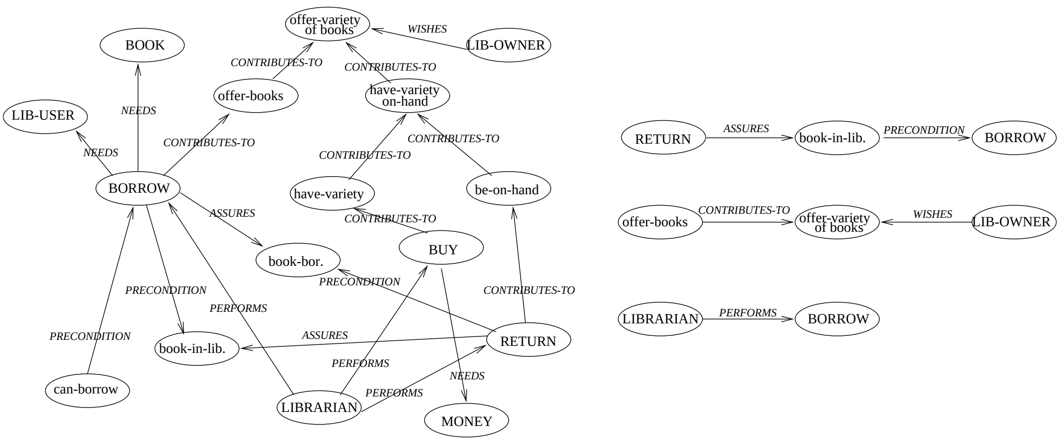

Figure 4.2: A part of the conceptual representation of a library management system and simple structures taken from it.

We consider a structure as any related set of nodes. For example, in figure 4.2, at left, we can see a piece of what could be considered a specification of a library management system. The overall set of nodes in conjunction with the relations that link them can be considered a complex structure; and it can be decomposed into more simple structures like those appearing at the right side of the figure. As those structures and possibly a taxonomy of concepts is all we have, we must rely on them to construct the knowledge model we will use for validation.

We will pay attention on those structures or groups of structures that are supposed to produce special kinds of properties in objects having them. Those structures will be captured in the form of concepts. That is, the extension of those concepts will be the set of objects having the corresponding structure.

In this way, structural properties are the basic element over which our hierarchy of concepts will be build. The generated concepts will be defined using combinations of *conceptualized structural chains*. A structural chain is a linear structural property; that is, an ordered set of nodes that are linked one to the next (except the last one) through a relation. A conceptualized structure is a structure in which the original objects that take part in the structure have been replaced with the concepts that have them as instances. Figure 4.3 shows the process of structure conceptualization through the examples in figure 4.2.

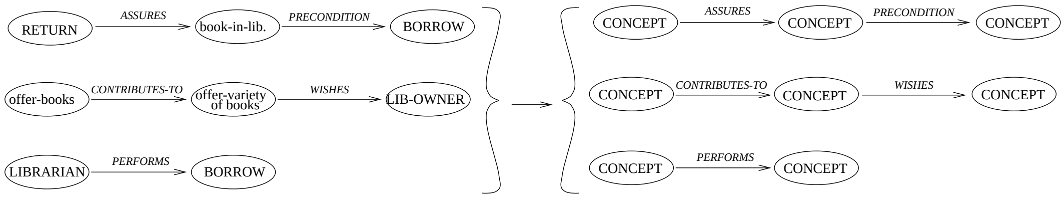

Figure 4.3: Structure conceptualization examples. Objects in the structural chains at the left are replaced by the classes they are instances of (for simplicity we suppose our initial hierarchy has an only concept: CONCEPT, and every object is an instance of it). This results in the conceptualized structural chains at the right. If an object is direct instance of more than one class, the conceptualization of a structural chain will result also in more than one conceptualized structural chain.

The main reason to conceptualize a structure is that the knowledge we want to learn (remember that we want to learn a model for a set of objects) is at the level of concepts and not at the level of instances. Structure conceptualization is what allows us to benefit from classification and generalization mechanisms. The use of classification mechanism is evident when replacing objects by the corresponding concepts. Concepts and instances are classified in a similar way they are in terminological logic systems such as KL-ONE or LOOM. With regard to generalization, structures are classified taking into account where the concepts are in the hierarchy. In this way, some structures can be more general than others because the concepts that contain are also more general (see figure 4.4).

To build more complex concept definitions than those allowed by simple structural chains we use simple logical connectives over them. The connectives allowed are *and*, *or* and *not*. In this way, concepts may have complex definitions in which structural chains are the basic element. The knowledge base taxonomy is determined by concepts definition.

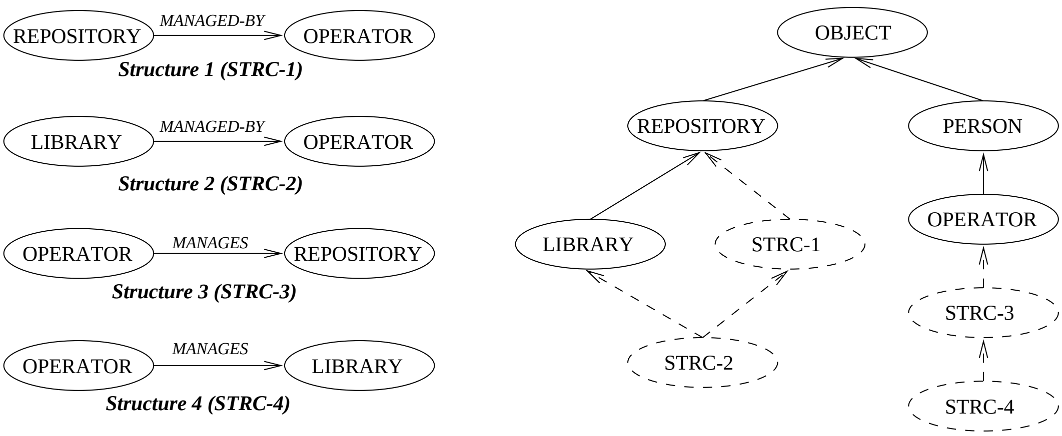

Figure 4.4: Structure classification. To the left, the structures we are going to create. To the right, the result of classifying the structures (dashed lines) in the original hierarchy (solid lines). STRC-1 and STRC-3 are classified as superclasses of STRC-2 and SCTR-4 respectively because REPOSITORY is more general than LIBRARY.

Once a concept has been generated (and also for concepts previously in the knowledge base, the *primitive* ones), we learn the properties (structural properties) that their instances have. To learn these properties, every object in the specification examples is classified automatically in the hierarchy and the properties are abstracted to the description of concepts.

So, every concept in the hierarchy will have attached a set of structural properties. Among them, we will distinguish those properties that define the concept (*definitional properties*) and those properties that are associated to it and conform its description (*associated properties*).

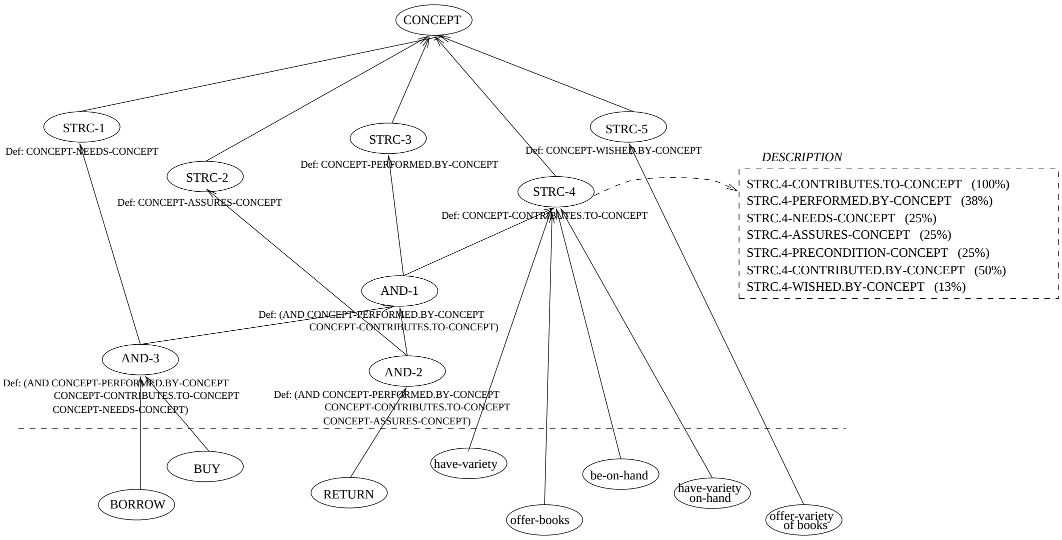

Figure 4.5: Simple concepts generated from properties taken from objects in the library example introduced before. In the figure we can see the concepts definition (below each concept), the objects that are instances of the concepts in the figure (below the dashed line) and the description for STRC-4. To make the example simpler, we have used only properties of length 1 (an only relation) and a previous conceptual model with an only concept (CONCEPT).

The first group establishes a double implication between the concept and its instances. The fact that an object accomplishes the definitional properties of a concept is a necessary and sufficient condition for this object to be an instance of the concept. This will be very useful for the automatic classification of objects in the taxonomy.

The second group represents structural properties that instances of a concept normally have; but they do not establish any kind of implication. These properties will be useful to detect relations among different structural properties. We use a probabilistic representation similar to that of COBWEB [18] for them. This allows to keep the ratio of instances that have every relevant property. To maintain complexity within an acceptable range, the learning system uses heuristics in order to determine which properties are relevant and which ones are not.

4.2 Knowledge and Metaknowledge

We have talked a lot about grouping objects into concepts, but we have still not mentioned what kind of knowledge must be learned. Typically, a specification contains two different kinds of information.

We find information about concrete things (tokens, those that conform the instance level in [14]) that decribes reality. In this way, a library management system (LMS) specification can talk about the "Barcelona Computer Science Faculty library". And we find also domain concept descriptions (conforming the domain level in [14]). In a LMS specification we will probably find descriptions of library, book, user, etc.

Al the representation level, the classification mechanism allows us to distinguish between these two levels of information and allows us to go further. If our representation formalism is flexible enough to work with concepts in the same way it works with objects, it can represent concepts as other concepts instances. This is used by systems such as TELOS [29] and KAOS [14, 36] to create a meta-knowledge level. Information conforming this meta-level constitutes a meta-model that allows us to impose certain restrictions over the domain level in the same way it does with the level of tokens. So, the meta-level is a powerful tool to find incompletenesses and inconsistencies in the domain level.

In our learning system we must be able to learn from the tokens and from the domain descriptions. We will have also these three levels: the *token level*, the *domain knowledge level* and the *meta-level*[2]

We can distinguish two learning levels: learning domain knowledge (from tokens) and learning the meta-model (from domain descriptions). We can decide to use one, the other or both of them. Of course *meta-learning* will be more useful than *domain-learning* because it will be useful for every domain and because most part of a specification is constituted indeed by domain descriptions.

A bidirectional flux of information exists between the token level and the domain knowledge level [17] and between the domain knowledge level and the meta-level (see figure 4.6). Learning is responsible for the flux from a level to the immediate upper level and a supposed type system driven by the classification mechanism is responsible for the flux from a level to the immediate lower level.

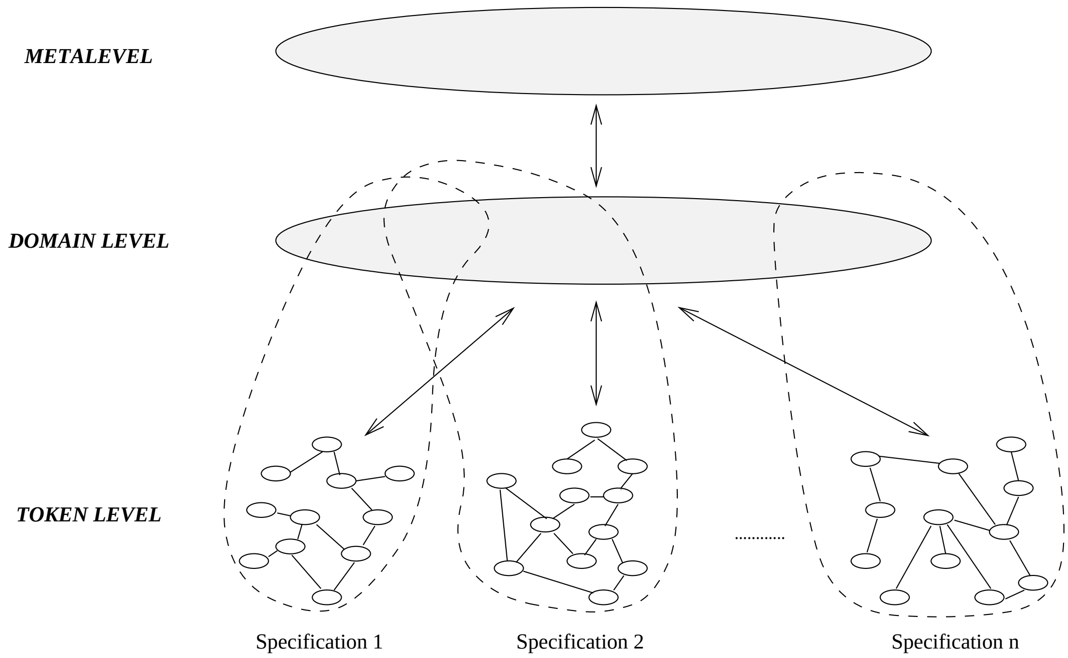

Figure 4.6: Bidirectional flux of information between different knowledge levels.

4.3 Concept Generation

But, what is the overall learning process really doing? From an inductive learning point of view, it is trying to learn definitions for structural properties. It defines concepts according to a set of structural properties (probably these concept definitions do not match property definitions). When a concept is generated, its description (a probabilistic representation) is built from the instances conforming the concept extension. For every concept we will obtain a list of structural properties and the ratio of concept instances that have them (always 1 for definitional properties).

[2]Although [29] states that we can have as many levels of knowledge as we want, those really useful for us are the first three. What we want is to model the information in the specification. As we have seen, this information belongs to the first two levels. The model of a level is given by its immediately upper level. So, adding a third level (we do not need to model it, we only need to use it) is enough.

Thus, for every structural property we will have a probabilistic distribution over all the concepts in the knowledge base. As every one of these concepts will have a definition, we will have obtained a definition for structural properties.

So, concept definitions are the basic elements to perform induction. And the definitions corresponding to the concepts in the knowledge base determine the *generalization language* (see [27]).

As any possible structure can be expressed with structural chains and the use of the allowed connectives, our representation system allows a generalization language expressive enough. The bias necessary to perform useful inductive leaps [27] will be provided by the *concept generation mechanism*.

This mechanism is based in *concept utility* (an heuristic measure that tries to catch the usefulness of a concept) and a set of *operators* that generate concepts and are triggered under certain situations.

Concept utility is similar to COBWEB's category utility [18] but adapted to our environment. Different to COBWEB, that only has descriptions, our learning system has descriptions and definitions. We can only compare concepts with comparable definitions. Two definitions are comparable when one is more specific than the other. So, for the utility of a concept we will only take into account those concepts that are more general or more specific than it. As we will compare only parents with sons, the *intra-concept similarity* (or intra-class) measure has little sense; so, we will use only the *inter-concept dissimilarity* measure.

Dissimilarity between two concepts is computed using common information in their descriptions. Lack of information is penalized. This lack of information is measured as the ratio between the properties a description has and the total of properties it could have. Because the penalty, and different to other approaches such as [35], the dissimilarity measure is not really a distance but only an heuristic (dissimilarity between an object and itself could be greater than 0 because of lack of information).

In what refers to operators, they constitute the basic tool that allows us to generate new concepts. We have three different kinds of operators:

instance operators generate concepts from the new instances introduced into the knowledge base. With them we can generate concepts corresponding to the simple structures present in the instances.

concept operators generate concepts from the definition of other concepts. They work by modifying directly the definition of a given concept and producing another definition. For example, the *generalize-chain operator* modifies a definition by replacing nodes in a structural chain by more general nodes (either concepts or relations). In this way, in figure 4.4 we can obtain STRC-1 and STRC-3 from STRC-2 and STRC-4 respectively. This kind of operators is triggered when a new concept has been generated and introduced into the knowledge base.

concept set operators Generate concepts from a group of concepts. They can be triggered at any moment while in the learning process. Triggering conditions should be true when the operator is most suitable to be applied. At the present moment we only have one concept set operator: the *clustering operator*; and it is most suitable to be applied at the end of the learning process, when all concepts except those generated directly or indirectly by the operator itself have been generated. This operator allows to build generic concepts from more specific concepts with similar descriptions; and in this way it tries to reach property definitions.

When a concept is generated by an operator it is not created into the knowledge base but its definition is introduced into a *waiting list*. The operator is also responsible for computing an a *priori concept utility* for the definition using the information present in the knowledge base. Then, the concept generation mechanism has a criterion for selecting definitions from the waiting list and creating the corresponding concepts in the knowledge base. This criterion can be a simple one, such as to compare the a priori concept utility to a threshold value; or a more complex one (it could take into account the number of concepts that have been generated, the simplicity of the definition, etc).

4.4 Relation Learning

In addition to the taxonomy of concepts, the knowledge base will have also a taxonomy for relations. This taxonomy is very useful in the learning process. To exemplify this, let's suppose that a set of concepts with simple definitions (a structural chain having only one relation) have the same description. Then, this description must be learned for every one of these concepts. If we use the relation taxonomy and the relations that appear in concepts definitions have a common parent, only a simple concept description should be learned (the one for the concept having the common parent relation in the structural chain that defines it).

For example, objects with unique identifier could have certain desirable properties (conforming a part of its description) that objects lacking it could not have. But *unique-identifier* will not appear directly in object descriptions taken from specifications. Instead, it could appear: serial number, ISBN, ISSN, bar code, etc. So, the taxonomy would help a lot here.

But, to be realistic, we cannot expect to have a complete relation hierarchy from scratch. As we want to take real benefit from relation hierarchies, we will use learning to complete or generate it. To perform learning over relations we take the same approach we have taken for concepts. As them, relations will also have a definition and a description.

Relation definitions are simpler than concept ones. Their basic element is not a structural chain but only primitive relations (those present in the knowledge base before learning started). Logical connectives that can be used are the same ones that are used for concepts. So, a unique identifier definition could be *(or serial-number, isbn, issn, bar-code, ...)*.

The description of a relation is more complex and is defined as the difference between the description of the most general concepts that have all their instances linked by that relation to other objects, and the description of the direct parents of those general concepts. This difference is a numerical subtraction computed from probabilistic representations of the concepts.

This seems a good heuristic to measure the properties induced in objects by the fact that they are linked to other objects by a given relation. In an intuitive manner, when a property has an associated positive value in a relation description, the presence of a link corresponding to that relation in an object helps in the presence of the property. Instead, when the value is negative, the presence of the corresponding link helps in the absence of the property. And finally, when it is zero, the link has no effect in the presence of the property.

4.5 Specification Validation

Once the learning process has finished, the system is ready to start validating specifications. As it has been said before, we see a specification as a set of object descriptions. Then, every description in the specification to be validated is classified within the learned hierarchy and its properties are compared to those of the concept (or concepts) that have it as an instance. Our problem has changed to that of validating a set of descriptions. Definitional properties will be useful to classify objects while associated properties will be useful to verify that a description is complete and consistent.

4.5.1 Completeness

When a description to be validated has been classified under a concept and a high ratio of instances of it have an associated structural property that the description has not, a potential incompleteness will have been detected.

What we are doing is to detect correlations between structures; if a group of objects having all of them a common structure (the definitional properties) have also (all, or most of them) another structural property (an associated property) and the description we are validating hasn't it but has the definitional properties, we could suspect that the description being validated is incomplete. The evidence ratio we have to state it will depend on the ratio of the instances of the concept that have the associated property. Figure 4.7 shows an example of this kind of check. We can see that although the associated structural property does not says which information the specification lacks (this would be impossible), it gives a direct hint for the software engineer to find it out.

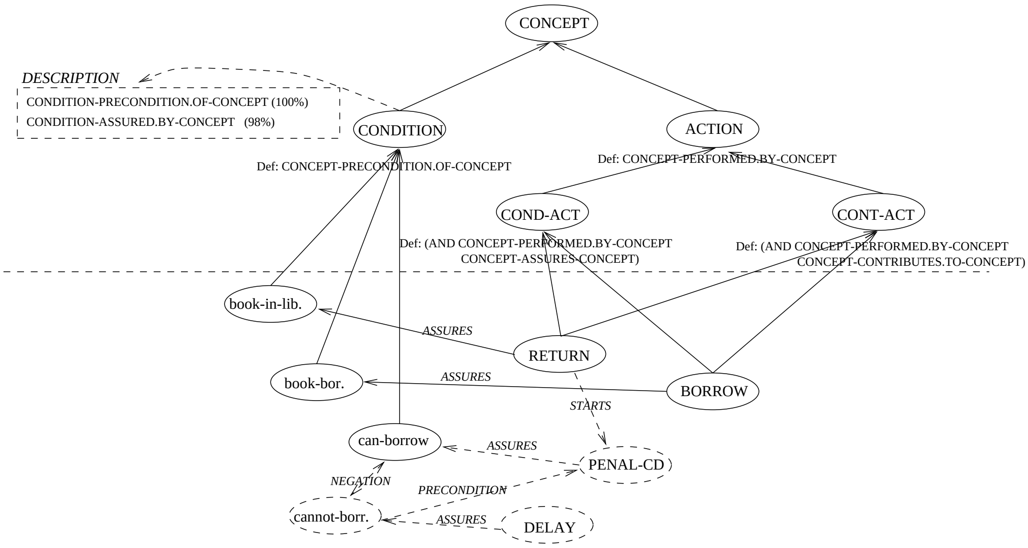

Figure 4.7: Validation of the 'can-borrow' condition results in the system warning that the condition is not assured by any action because 98action. The software engineer adds some piece of knowledge to fix this error (dashed line).

But, what happens if the description of an object is so incomplete that it is not classified under the right concept? The object will be classified under an ancestor of it that maybe lacks some associated properties that the right concept would use to detect incompletenesses. To solve this we can iterate the classification-validation process. Once we have corrected some incompletenesses, the concept is classified again. New incompletenesses will be found and so on until no new incompletenesses are found.

In a complementary way, if the description to be validated has been classified under a concept with no direct instances it is probable that this concept is an abstract one and it is not the right concept. If the abstract concept is so abstract that it does not allow us to start the iterative process explained in the preceding paragraph, we can try to use definitional properties of the sons of the concept.

First we look for all the partitions of the abstract concept[3]. Then, as the object corresponding to the description must be in all the partitions, properties defining every partition (the union of properties defining concepts in them) and not present in the parent's definition are proposed by the system as possible ways to complete the description.

Figure 4.8 shows the process of completing the description of a goal. As *offer-variety-of-books* does not contributes to another goal nor is wished by any agent, it cannot be classified neither as a *subgoal* neither as a *main goal*. But as we know that the concepts *subgoal* and *main goal* constitute a partition of *goal* (we have learned it empirically from the set of valid specifications we have used in the learning process), we know that *offer-variety-of-books* should be an instance of *subgoal* or an instance of *main goal*. The validation system reports a warning and the software engineer decides *offer-variety-of-books* was a *main goal* and adds the corresponding knowledge.

4.5.2 Consistency

Similar to completeness checks, when a description to be validated has some structural property that is not present in a high ratio of the instances of the concepts that have it as an instance, we

[3] We understand a partition of a concept as a subset of its direct subconcepts that covers completely the concept. That is, every instance of the concept is also an instance of one of the subconcepts in the partition.

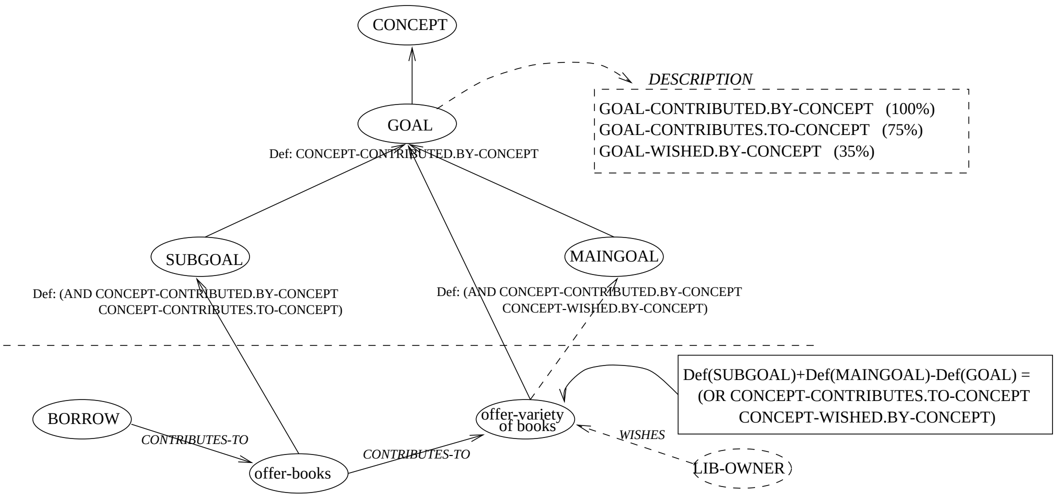

Figure 4.8:

may suspect the existence of a potential inconsistency or the existence of a piece of unnecessary information.

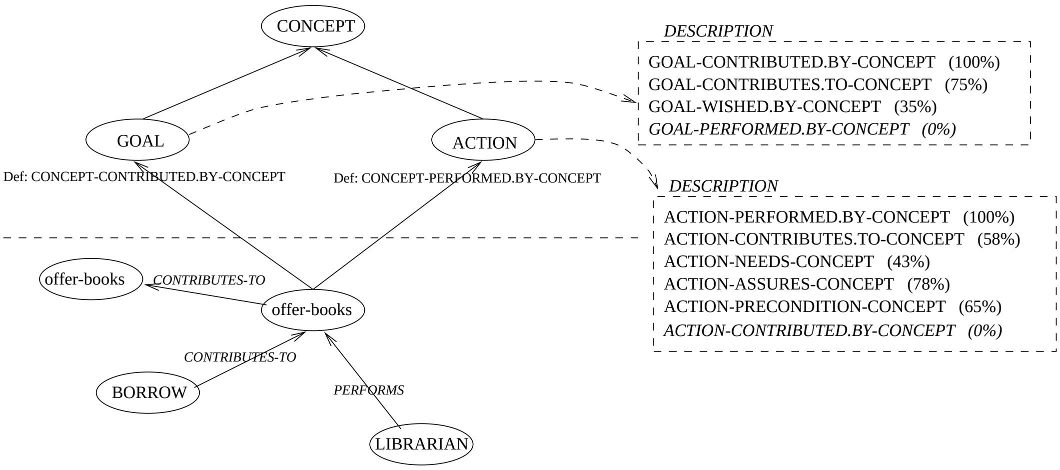

Figure 4.9: Example of an inconsistency that can be discovered by the system.

In figure 4.9 we can see a bad specification of a goal. In fact, the specifier has mixed a goal with an action. Then, the object *offer-books* is classified as a *goal* and as an *action*. As these two concepts have no common instances (except *offer-books*), the system reports a possible inconsistency for *offer-books*, stating that some of the structural properties in the definition of *gola* or *action* should not be in *offer-books*.

4.6 Training Set

The election of the exemplars constituting the training set is extremely important for a machine learning method that learns from examples. In this work, the training set is constituted by specifications. In this section we explain the properties the specifications must hold for an effective learning and validation.

4.6.1 Representativity

First of all, specifications must be *fully representative* of the domain we are involved with. If a part of the domain is not represented in the training set, nothing will be learned about it. This can result in two distinct errors. If the systems detects the lack of a structure that has always been present in the training set examples, an incompleteness will be reported. By the other hand, when a new structure is found, the validation system will report an inconsistency.

It is evident we cannot pretend every possible situation to be present in the training set. We cannot be sure about incompletenesses and inconsistencies detected by the validation process. So, the ultimate decision about errors must be that of the software engineer [33]. Errors reported by the validation system must be understood as warnings about possible or probable errors. The software engineer must decide whether they are real errors or not.

All this does not mean that the work explained here is no worth. After a broad analysis of software engineering problems, [26] states that most of them belong to a tractable set of hierarchical classes. Then, if abstractions done by the learning mechanism are good enough, the number of new situations a specification may have respect to a set of them (the training one) decreases quickly when set size increases.

Moreover, when a specification has been validated, it can be used by the learning mechanism to refine the knowledge base. In this way, we can reduce the number of erroneous warnings at the time specifications become validated.

4.6.2 Consistency

It is very important for the specifications in the training set to be consistent. Otherwise, inconsistencies from the examples will be learned and the validation process will be useless.

4.6.3 Minimality

Specifications in the training set must also be minimal. In this way, we can be sure that every learned structural property is strictly necessary.

This is specially important for completeness checks. If learned properties are necessary, when a description lacks one of them we can say that description is incomplete. If training set specifications were not minimal, maybe some not necessary properties would be learned and the validation system wouldn't be able to decide whether the lack of a structure is an incompleteness or not.

4.6.4 Completeness

In the same way, training set specifications must also be complete. Otherwise, incomplete descriptions would be learned and some incompletenesses would not be detected by the validation system.

By the other hand, inconsistency checks would also be affected as some necessary properties would not be learned. This would produce inconsistency warnings when the validation system detects those unlearned properties.

4.6.5 Condition Weakening

The fact that we learn the ratio in which a concept (its instances) has a structural property makes the learning mechanism more robust and resistant to errors in the training set.

So, the properties explained in the previous paragraphs can be a little weakened. The learning mechanism does not need specification to be submitted to a special validation process in order to belong to the training set. Any specification that has been used in real software engineering problems is good enough. But, of course, the more errors the specifications in the training set contain the more examples will be needed for a reliable validation.

Chapter 5

Conclusion and Future Work

This paper explains a system that is able to construct a knowledge model from a set of good specifications. This model is constructed basically from structural information and it is used to validate other specifications.

Manual construction of the part of the knowledge base that will allow us to validate specifications (as we proposed in [5]) is a really tedious and difficult labor. The reasons are mainly three: problem complexity, the dependence of completeness and consistency definitions on the domain and other context factors; and the informality introduced by natural language. So, using machine learning techniques to construct such a knowledge base seems a good solution.

At the moment of writing this paper, we have tested our learning system with knowledge generated randomly from a meta-model similar to that of KAOS[1]; and the result is quite acceptable. The learning system has used only four operators (generalize chain, clustering and two instance operators). Relation learning hasn't been implemented yet.

In order to validate our work, real specifications must be used for the learning and validation process. A deeper study of machine learning for heuristic concept generation is also needed. It can also be interesting (and necessary) to study the possibility to remove unnecessary information that has been heuristically generated (concept utility may change dynamically when new concepts are created into the knowledge base). Finally, increasing the expressive power of the structural representation mechanism could also be interesting. In this way, concepts referring the structure itself could be introduced (for example SELF-nth, as referring the nth element in the structure). This would allow to learn more complex reasonings such as those typical from analogical reasoning.

[1] The meta-model allows us to generate, in a random manner, information conforming to it. This information plays the role of specifications in the training set. The meta-model is removed from the semantic network before the learning process starts. Then, at the end of the learning process we must have acquired the same knowledge provided by the meta-model.

Bibliography

[1] AECMA. *AECMA Simplified English, A Guide for the Preparation of Aircraft Maintenance Documentation in the International Aerospace Maintenance Language*, 1989.

[2] J. Alvarez. An approach to the control of completeness in LESD. In *Proceedings of Doctoral Consortium of 2nd International Symposium on Requirements Engineering*, March 1995.

[3] J. Alvarez and N. Castell. An approach to the control of completeness based on metaknowledge. Technical Report LSI-94-50-R, Universitat Politècnica de Catalunya, 1994.

[4] J. Alvarez and N. Castell. Uso de técnicas de aprendizaje para la validación de especificaciones. *Novática*, (118):45–52, Noviembre–Diciembre 1995.

[5] J. Alvarez, N. Castell, and O. Slavkova. Combining knowledge and metrics to control software quality factors. In *IFIP International Conference on Achieving Quality in Software (AQUIS'96)*, Florence, January 1996.

[6] V. Basili and B. Perricone. Software errors and complexity: An empyrical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[7] A. Borgida. Conceptual modeling of information systems. In Brodie and Mylopoulos [11], chapter 31, pages 461–469.

[8] A. Borgida, S. Greenspan, and J. Mylopoulos. Knowledge representation as the basis for requirements specification. *IEEE Computer*, 18(4):82–91, April 1985.

[9] M. Borillo, Y. Toussaint, and A. Borillo. Motivations du projet LESD. In *Linguistic Engineering Conference*, Versalles, France, January 1991.

[10] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, August 1985.

[11] M. Brodie and J. Mylopoulos, editors. *On Knowledge Base Management Systems*. Springer-Verlag, 1986.

[12] N. Castell, O. Slavkova, Y. Toussaint, and A. Tuells. Quality control of software specifications written in natural language. In *Proceedings of 7th International Conference on Industrial & Engineering Applications of AI & Expert Systems (IEA-AIE'94)*, pages 37–44, Austin, Texas, June 1994. Gordon and Breach Science Publishers.

[13] D. Cordes and D. Carver. Evaluation method for user requirements document. *Information and Software Technology*, 31(4):181–188, May 1989.

[14] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[15] Alan M. Davis. *Software Requirements, Objects, Functions and States*. Prentice-Hall, 1993.

[16] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. Lassie: A knowledge-based software information system. *Communications of the ACM*, 34(5):35–49, May 1991.

[17] M.S. Feather. Requirements reconnoitering at the juncture of domain and instance. In Requirements Engineering [31], pages 73–76.

[18] D. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[19] S. Greenspan, A. Borgida, and J. Mylopoulos. A requirements modeling language and its logic. In Brodie and Mylopoulos [11], chapter 32, pages 471–502.

[20] S. Greenspan, J. Mylopoulos, and A. Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of 16th International Conference on Software Engineering*, pages 135–147, Sorrento, Italy, 1994.

[21] IEEE. *IEEE Guide to Software Requirements Specification*, IEEE-830, 1984.

[22] M. Jackson. *System Development*. Prentice-Hall, 1983.

[23] M. Jackson. Problems and requirements. In Requirements Engineering [32], pages 2–8.

[24] M. Jarke, J. Bubenko, C Rolland, A. Sutcliffe, and Y. Vassiliou. Theories underlying requirements engineering: An overview of nature at genesis. In Requirements Engineering [31], pages 19–31.

[25] R. MacGregor. Using a description classifier to enhance deductive inference. In *Proceedings of the 7th IEEE Conference on Artificial Intelligence*. IEEE CS Press, 1991.

[26] N.A.M. Maiden, P. Mistry, and A.G. Sutcliffe. How people categorise requirements for reuse: a natural approach. In Requirements Engineering [32], pages 194–203.

[27] T.M. Mitchell. Generalization as search. In J. Shavlik and T. Dietterich, editors, *Readings in Machine Learning*, pages 96–107. Morgan Kaufman Publishers, 1990.

[28] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, June 1992.

[29] J. Mylopoulos, M. Jarke, A. Borgida, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[30] R.S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, NY, USA, third edition, 1992.

[31] *Proceedings of 1st International Symposium on Requirements Engineering*, San Diego, California, January 1993. IEEE CS Press.

[32] *Proceedings of 2nd International Symposium on Requirements Engineering*, York, England, March 1995. IEEE CS Press.

[33] H. Reubenstein and R. Waters. The requirements apprentice: Automated assistance for requirements acquisition. *IEEE Transactions on Software Engineering*, 17(3):226–240, March 1991.

[34] O. Slavkova. Modelo para el control de calidad en LESD basado en medición del software. Technical Report LSI-93-26-R, Universitat Politècnica de Catalunya, 1993.

[35] G. Spanoudakis and P. Constantopoulos. Similarity for analogical software reuse: A computational model. In *Proceedings of 11th European Conference on Artificial Intelligence*, Amsterdam, August 1994.

[36] A. van Lamsweerde, R. Darimon, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In Requirements Engineering [32], pages 194–203.