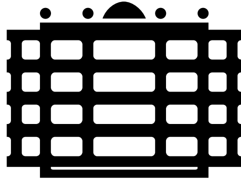

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-08-02

Using Semantic Web Technology in Requirements Specifications

Petr Kroha · José Emilio Labra Gayo

November 2008

Chemnitzer Informatik-Berichte

Contents

1 Introduction **7**
1.1 The Motivation of Checking Requirements 7
1.2 Ontology as a Tool for Checking Requirements 8

2 Ontologies and their Properties **11**
2.1 Ontology Writing Language OWL – three versions 11
2.1.1 Class in Ontology 11
2.1.2 Property 11
2.1.3 Constraints 12
2.1.4 Object Identity 12
2.2 Ontology Design and Object-oriented Design 12

3 Implementation of an Ontology **15**
3.1 Using Protégé for Building an Ontology 15

4 Features of Requirements that can be Modeled in UML but not in OWL **16**
4.1 Behavioral Features 16
4.2 Complex objects 16
4.3 Access control 17
4.4 Keywords 17

5 Converting UML to OWL **18**
5.1 Existing Convertors 18

6 Description Logics and Reasoning in Ontologies **19**
6.1 Introduction 19
6.2 Basic definitions and foundations 20
6.3 Using Description Logics for Data Modeling 21
6.3.1 Intesional and extensional approach 21
6.4 The Syntax and Semantics of DLs 23
6.4.1 The Logic of Descriptions 24
6.4.2 Reasoning with DLs 24
6.4.3 Descriptions as constraints 25
6.4.4 DLs for stating rules 25
6.4.5 On the generality of the DL framework 25
6.4.6 Complexity vs. expressiveness 26

7 Existing Reasoners in Ontologies **27**
7.1 Using RacerPro for Reasoning in Ontology 27
7.2 Using Jess for Reasoning in Ontology 27

8 The Proposed Concept **29**
8.1 The existing tool TESSI
Architecture and Dataflow 29
8.2 The proposed extensions of the tool TESSI 30

9 Requirements Engineering **34**
9.1 Domain Knowledge and Project Requirements 34
9.2 Requirements Elicitation 36
9.3 Requirements Analysis 37

10 Related Work to Checking Requirements Specification **39**
10.1 Checking Unambiguity of
Requirements Specification 39
10.1.1 Linguistic Ambiguities 40
10.1.2 Requirements-specific Ambiguities 41
10.1.3 An Approach based on Neuro-Lingustic Programming 41
10.2 Checking Inconsistency of Requirements Specification 48
10.2.1 The Inconsistency Management Process 50
10.2.2 Human Inspection 51
10.2.3 Similarity Analysis 51
10.3 Checking Completeness of Requirements Specification 52
10.3.1 Logical Completeness 52
10.3.2 Computational Completeness 52
10.3.3 Completeness of Ontology 52
10.3.4 Completeness of Requirements 53
10.4 Checking Validity of Requirements Specification 53
10.5 Why to Use Ontology for Checking Requirements Specifications 54
10.5.1 Using Shared Ontologies in Knowledge Acquisition 55
10.6 Weak points of related papers 55

List of Figures

8.1 Architecture and dataflow of the existing tool TESSI 30
8.2 Architecture and dataflow of the proposed tool TESSI 31
8.3 Checking requirements process 33

10.1 Inconsistence 49

In this report, we investigate how the methods developed for using in Semantic Web technology could be used in capturing, modeling, developing, checking, and validating of requirements specifications.

Requirements specification is a complex and time-consuming process. The goal is to describe exactly what the user wants and needs before the next phase of the software development cycle will be started. Any failure and mistake in requirements specification is very expensive because it causes the development of software parts that are not compatible with the real needs of the user and must be reworked later.

When the analysis phase of a project starts, analysts have to discuss the problem to be solved with the customer (users, domain experts) and then write the requirements found in form of a textual description. This is a form the customer can understand. However, any textual description of requirements can be (and usually is) incorrect, incomplete, ambiguous, and inconsistent.

Later on, the analyst specifies a UML model based on the requirements description written by himself before. However, users and domain experts cannot validate the UML model as most of them do not understand (semi-)formal languages such as UML.

It is well-known that the most expensive failures in software projects have their roots in requirements specifications. Misunderstanding between analysts, experts, users, and customers (stakeholders) is very common and brings projects over budget.

The goal of this investigation is to do some (at least partial) checking and validation of the UML model using a predefined domain-specific ontology in OWL, and to process some checking using the assertions in descriptive logic.

As we described in our previous papers, we have implemented a tool obtaining a modul (a computer linguistic component) that can generate a text of requirements description using information from UML models, so that the stakeholders can read it and decide whether the analyst's understanding is right or how different it is from their own one.

We argue that the feedback caused by the UML model checking (by ontologies and OWL DL reasoning) can have an important impact on the quality of the outgoing requirements.

This report contains a description and explanation of methods developed and used in Semantic Web Technology and a proposed concept for their use in requirements specification. It has been written during my sabbatical in Oviedo and it should serve as a starting point for theses of our students who will implement ideas described here and run some experiments concerning the efficiency of the proposed method.

1 Introduction

1.1 The Motivation of Checking Requirements

Any system that is designed to solve a customer problem must be evolved from a set of functional requirements that constitute a functional specification. There are also non-functional requirements, e.g. performance, security, that result in non-functional specification but at the very moment we focus on functional requirements in this report.
In this context we have to explain the following terms:

- customer - somebody who pays us for the delivered system,
- user - somebody who will use the system,
- domain expert - somebody who understands all details of the problem domain,
- client - denotes customer or user,
- stakeholder - denotes all objects above.

In the following text, we write mostly about the requirements process between an analyst on the one side and user on the other side, even though all stakeholders are more or less involved. But for the purpose of this text we can simplify in most cases.

The requirements are most often elicited from the user by the analyst through the use of some type of an interview [134]. The root of the requirements problems lies in the common ground between the user and the analyst, which can only be discovered through communication activities that facilitate a sharing of information [35].

The most expensive failures in software projects have their roots in requirements specifications. Misunderstanding between analysts, domain experts, users, and customers is very common. They may have the following reasons:

- Informal conversation at the beginning

  Every project starts with a conversation between the analysts und users concerning the needs and wishes of the users. This conversation and subsequent conversations are informal and not precise but they are critical to the success of the project.

- Different level of understanding and vocabularies

  Each participant brings his own experiences and knowledge to bear on the situation but is unversed in the language of the other. This level of unfamiliarity requires each participant to make conscious thoughts and learned experiences. Sometimes there are pieces of knowledge that will not be explicitly said, remaining in mind and only later identified when the finished product reveals their omission.

- Self-evident features

  Similarly, there will be notions that simply are not remembered and therefore never contributed to the conversation. Analysts never have heard from them and the users consider them for self-evident. Several false assumptions and biases serve to constrain the correct understanding.

- Private interests of the client

  The conversation is often charged by a subjective focus on the problem. The users do not want some changes in the system to happen because they feel possible disadvantages for themselfs.

1.2 Ontology as a Tool for Checking Requirements

Requirements are based on knowledge of domain experts and users' needs and wishes. One possible way to classify this knowledge and then fashion it into a tool is through ontology engineering. This way will be discussed in this report.

Ontologies can be seen as explicit formal specifications of the terms in the domain and relationships among them. They care for a shared understanding of some domain of interest [136]. Such an understanding can serve as the basis for communication in requirements development. Ontologies are a guarantee of consistency [67] and enable reasoning. An ontology-based requirements specification tool may help to reduce misunderstanding, missed information, and help to overcome some of the barriers that make successful acquisition of requirements so difficult.

Simplified, ontologies are structured vocabularies having the possibility of reasoning. It includes definitions of basic concepts in the domain and relations among them. It is important that the definitions are machine-interpretable and can be processed by algorithms.

Why would someone want to develop an ontology?
Some of the reasons are:

- To share common understanding of the structure of information among people or software agents.
- To enable reuse of domain knowledge.
- To make domain assumptions explicit.
- To separate domain knowledge from the operational knowledge.
- To analyze domain knowledge.

Ontology research has primarily focused on the act of engineering ontologies or it has been explored for use in domains other than requirements elicitation, specification, checking, and validation.

In ontology-based requirements engineering, the correctness, completeness, consistency and unambiguity of ontology should be guaranteed so far that it can be used to guide the requirements elicitation and requirements evolution. As we will show below, the guaranty is difficult, especially the guaranty of completeness. The requirements evolution will often be forgotten, but the never-ending changing of environment causes that requirements change and then software systems have to be changed constantly. In this context the traceability of requirements evolution and the traceability of requirements implementation is very important.

The goal is not only to develop requirements specification but to create (or to have available) a domain ontology in every project before the requirements specification process will be started. Software houses are usually specialized on producing software systems that solve a specific set of problems, e.g. information systems for financial institutions. Therefore, the objective is to have an ontology domain available for a given field of applications and to check requirements of all projects being developed for this field.

For an ontology being successfully used in requirements checking, it has to have the following properties: completeness, correctness, consistency, and unambiguity.

The intuitive meaning is:

- correctness means that the knowledge in the ontology does not violate the domain rules that correctly represent the reality,
- consistency means that there are no contradictory definitions in ontology,
- completeness means that the knowledge in ontology describes all aspects of the domain,
- unambiguity means that the ontology has defined a unique or unambiguous terminology. There are not obscure definitions of ontology concepts, i.e. each entity is denoted by only one, unique name, all names are clearly defined and have the same meaning for the analyst and all stakeholders.

Correctness and consistency are logical properties that can be checked by some reasoning mechanism provided that this mechanism works correctly.

This is what we can use for improving the quality of requirements. After we have checked the correctness and consistency of the corresponding domain ontology (domain knowlegde), we can check whether the requirements (transformed into an ontology) are correct and consistent mutually, and correct and consistent to the given domain ontology.

Completeness is a complex concept. We know about other kinds of completeness that are related to computing. For example, logical completeness is a property associated with combining a procedure for constructing well-formed formulas, a definition of truth that relates to interpretations and models of logical systems, and a proof procedure that allows new well-formed formulas to be derived from old well-formed formulas. A logical system is logically complete if every true well-formed formula can be derived.

Consistency is the other side to logical completeness. In inconsistent systems falsity can be derived because of an contradiction and any well-formed formula can be derived as true. This is sure not what we want.

A second kind of completeness is computational completeness, which is a property of a programming language. Consider the programming language called a Turing machine. No one has ever invented a programming language in which a program could be written that could not be represented as a Turing machine. Indeed, the Church-Turing thesis is that it is impossible to have a programming language more powerful than a Turing machine. This thesis has not been proved, but it has never been seriously challenged. Programming languages exist that are not complete (e.g. SQL). In other words, it is possible to formulate programs that these languages cannot represent. For example, SQL does not support recursion, which makes it unsuitable for a range of problems including bill-of-materials applications.

Both, logical completeness and computational completeness are properties of the languages and reasoning platforms used to express statements, not properties of the statements themselves. We would therefore expect that ontological completeness would be a property of the language and reasoning platform used to specification of the domain. The task of ontology is to categorise

all possible aspects of the reality (i.e. all aspects relevant for all applications we suppose to do by help of the ontology) and all rules on how they can be changed.

This potential for completeness, which as we have seen above is analogous to logical and computational completeness, has been called ontological adequacy by Guarino [70]. It is, of course, impossible to determine a priori whether an instance of an information system (a specialised ontology) actually completely matches the domain, because that judgment depends on how successfully the ontology and the reality interact. At this point we have to state that the problem of ontology completeness is too complex and it is out of scope of our report. We will only use the intuitive description given above.

We cannot be sure that our ontology is complete but we can suppose that it is close to be complete if it has been used successfuly in many applications. The situation is similar to the problem of library package verification. Our goal is to use an ontology in requirements engineering so we have to define what completeness of requirements means.
Completeness for requirements means that:

- all categories of requirements (functional and non-functional requirements) are addressed,
- all responsibilities allocated from higher-level specifications are recognized,
- all use cases, all scenarios, and all states are recognized,
- all assumptions and constraints are documented.

This is a good intuitive definition but it is not constructive. We cannot to use it to decide whether our requirements are complete.

The problem is in the ”all”, of course. What we can do is to hope that the domain ontology is ”more complete” than the developed requirements so that we can check the requirements by comparing them with the ontology and find (perhaps) that there are some aspects described in the ontology but not described in the requirements.

As we argued above we cannot check an ontology for completeness. But because of the possibility of reasoning, we can check an ontology to ensure the correctness and accuracy of it before using it to guide and check application requirements specification.

2 Ontologies and their Properties

As we have already mentioned above an important part of the proposed concept is the conversion of UML model to an ontology described in OWL. In this chapter we describe the part of the concept that concerns ontologies in detail.

2.1 Ontology Writing Language OWL – three versions

As we explain below, there are three versions of OWL (Ontology Writing Language) that have a different expressiveness and different computational complexity:

- OWL Lite - supports those users primarily needing a classification hierarchy and simple constraints. For example, while it supports cardinality constraints, it only permits cardinality values of 0 or 1.
- OWL DL - supports those users who want the maximum expresiveness while retaining computational completeness and decidability. OWL DL includes all OWL language constructs, but they can be used only under certain restrictions. For example, while a class may be a subclass of many classes, a class cannot be an instance of another class.
- OWL Full - is meant for users who want maximum expresiveness and the syntactic freedom of RDF with no computational guarantees.

2.1.1 Class in Ontology

A class is an entity that can have instances. It may be regarded as a sort or as a set of instances. Similarly as in object-oriented modeling those instances can change as new facts are introduced.

Classes can be related to each other by subclass and disjointness relationships. One can construct classes from other classes using union and intersection or explicitly enumerating all of its instances.

One operation that is problematic is the complement. For this to be well defined, one must postulate a universal class with the property that every entity that can be the instance of any class is an instance of the universal class. This is a strong assumption even if it is only assumed within a specified context. In practice, all that is really needed is a notion of a relative complement.

2.1.2 Property

One important difference between ontology languages and UML is the support for properties. UML associations are subsidiary to the classes that they relate. In ontology languages, on the other hand, one can introduce a property without any specific reference to any classes.

OWL distinguishes between two main categories of properties that an ontology builder may want to define:

- Object properties link individuals to individuals,

- Datatype properties link individuals to data values.

A property is:

- e.g., hasAccount, isProducedBy,
- an association to a range of values,
- an association to a domain of objects,
- a collection of relationships between individuals (and data),
- a way of describing a kind of relationship between individuals,
- an object in the world (in OWL Full).

An object property is defined as an instance of the built-in OWL class owl:ObjectProperty. A datatype property is defined as an instance of the built-in OWL class owl:DatatypeProperty.

As a result, properties are aspects that cross-cut classes. A single property will correspond to several UML binary associations. Properties of an ontology can be constrained in various ways (domain, range and cardinality constraints) with respect to specific classes. One can constrain a property to be a subproperty of another, and one can construct the inverse of a property.

2.1.3 Constraints

Properties can be constrained in various ways. There are global constraints that apply to the property as a whole, such as transitivity. There are other constraints that exist only within the context of a class. Constraints are usually given as axioms in predicate calculus.

2.1.4 Object Identity

Objects in ontologies have an object identity in the same way that objects have an identity in object-oriented systems. The one difference is that one can assert that two objects are the same, even though they might have been separately constructed. It is also possible for objects to be inferred to be equivalent. Because objects can be equivalent to each other, one must explicitly state that two objects are different if this is useful for purposes of inference.

2.2 Ontology Design and Object-oriented Design

Some ontology-design ideas are the same as in object-oriented design, e.g. in Booch [19]. However, ontology development is different from designing classes and relations in object-oriented programming.

Object-oriented programming primarily centers around methods in classes. A programmer makes design decisions based on the structural and operational properties of classes, whereas an ontology designer makes these decisions based only on the structural properties of a class hierarchy because functionality (methods) is not part of classes in ontology.

As a result, a class structure and relations among classes in an ontology are different, i.e. may be different, from the structure for a similar domain in an object-oriented program.

For the purposes of this report an ontology is a formal explicit description of:

- classes describing concepts in a domain of discourse,
- properties describing relations between concepts (i.e. specialization/generalization, aggregation, association),
- properties of each concept describing various attributes of the concept (sometimes called slots or roles),
- restrictions on slots (could be compared to constraints in object-oriented modeling).

An ontology together with a set of individual instances of classes (objects in object-oriented modeling) constitutes a knowledge base. The difference between an ontology and a knowledge base is not exactly defined.

To describe a domain means to develop an ontology describing the domain.

In practical terms, developing an ontology includes:

- defining classes in the ontology,
- arranging the classes in a taxonomic (subclass/superclass) hierarchy,
- defining slots (they correspond to attributes in object-oriented modeling),
- defining slot restrictions (domain and its constraints for each slot, i.e. allowed values for these slots),
- defining individual instances by filling in the values for slots.

One of the ways to determine the scope of the ontology is to sketch a list of questions that a knowledge base based on the ontology should be able to answer, so-called competency questions.

The same method is used for the design of an object-oriented information system that starts with queries which the system should be able to answer. The difference is that an object-oriented information system has its operational part stored in methods of classes that are invoked by messages given in the message-flow corresponding to the query. The answer can be any object and any change of the state that is made during the query processing.

An ontology has its operational part represented by reasoning and it is stored in rules of the reasoning system (and the inference machine behind) that tries to determine whether the query can be derived from facts and rules stored in the ontology. The answer is true or false.

Similarly as in requirements specification, we have to define (or at least to write down) a list of all terms we would like either to make statements about or to explain to a user. This means that we define a vocabulary used for a description of the domain.

Considering requirements specification, this vocabulary has to be expressive enough to describe not only the domain in the sense of the IS-state but also the domain in the sense of the PROPOSED-state. This means that we have to built an ontology for a description of the reality which includes also the results of the proposed system.

Compared with object-oriented modeling, we can use the word slot in modeling of ontology instead of the word attribute in object-oriented modeling and we have to notice that there are not methods to be modeled because ontologies describe static properties only.

The usual goal of constructing specialised ontologies has been to represent the semantics of the domain of discourse, exactly its part that can be described by an ontology.

Ever since the introduction of the entity-relationship model [30], the construction of specialised ontologies (often called conceptual models) has been an essential part of the information systems design process. Historically, these ontologies have been partial. They have not

provided a model of all phenomena of interest in the domain of discourse. Rather, they have been used to describe only those phenomena that can be represented conveniently in database schemas. Other phenomena (e.g. events and transformations) have been represented via programming language statements, initially using languages like COBOL.

Recently, specialised ontologies that are more complete have begun to appear. For example, workflow modeling grammars have now been developed that allow scripts to be generated which describe processes in the domain of discourse [125]. In conjunction with the scripts generated via conceptual modeling grammars, these scripts provide a more complete coverage of the phenomena of interest in the domain of discourse.

As mentioned above, it is necessary to stress again that ontology does not describe behavior, i.e. that we can check only the static part of the UML model.

3 Implementation of an Ontology

3.1 Using Protégé for Building an Ontology

To build an ontology, we need an ontology editor as an ontology development tool to use it for knowledge acquisition and as a knowledge-base framework. There are some of them but lately the Protégé platform developed in Stanford has become a standard in use. According to [116], the Protégé platform supports two main ways of modeling ontologies:

- The Protégé-Frames editor enables users to build and populate ontologies that are frame-based, in accordance with the Open Knowledge Base Connectivity protocol (OKBC). In this model, an ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concepts, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes - individual instances of the concepts that hold specific values for their properties.
- The Protégé-OWL editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL). "An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics. These entailments may be based on a single document or multiple distributed documents that have been combined using defined OWL mechanisms" (see the OWL Web Ontology Language Guide).

Protégé is a popular, modular ontology development and knowledge acquisition tool. It produces output (that is, ontologies and knowledge bases) that other programs can read. Protégé assumes that the systems performance engine runs as a separate program, perhaps on a different platform. This separation is often advantageous because the development and runtime environments are different. Nevertheless, tight integration with the performance engine sometimes is essential. Currently, we can access Protégé knowledge bases using Java and an application programming interface, but doing so is time consuming. Protégé ontologies can be exported into a variety of formats including RDF(S), OWL, and XML Schema. Protégé is based on Java, is extensible, and provides a plug-and-play environment that makes it a flexible base for rapid prototyping and application development.

4 Features of Requirements that can be Modeled in UML but not in OWL

A comparizon of UML 2.0 and OWL in details is given in [112]. Our goal is to derive an ontology in OWL (we can call it requirements ontology to make a difference to domain ontology) from requirements modeled in UML. So, we focus to properties contained in UML but not contained in OWL because the main question in our project is how the UML model must be constrained to make its transformation into an ontology in OWL possible.

4.1 Behavioral Features

UML allows the specification of behavioral features, which can be seen as operations (programs). One use of them is to calculate slot values.

Facilities of UML that support modeling programs include operations, which are:

- method names,
- responsibilities - specify which class is responsible for what action,
- static operations - operations attached to a class like static attributes,
- interface classes - specify interfaces to operations,
- abstract classes - their operations are specified in subclasses,
- qualified associations - programming language data structures,
- active classes - their instance controls its own thread of execution control.

It is proposed that the ODM (Ontology Definition Metamodel and OWL for its description) omit behavioral features of UML.

4.2 Complex objects

UML supports various flavors of the part-of relationship between classes. In general, a class (of parts) can have a part-of relationship with more than one class (of wholes). One flavor (composition) specifies that every instance of a given class (of parts) can be a part of at most one whole. Another (aggregation) specifies that instances of parts can be shared among instances of wholes.

Composite structures are runtime instances of classes collaborating via connections. They are used to hierarchically decompose a class into its internal structure which allows a complex objects to be broken down into parts. These diagrams extend the capabilities of class diagrams,

which do not specify how internal parts are organized within a containing class and have no direct means of specifying how interfaces of internal parts interact with its environment.

Ports and Connectors model how internal instances have to be organized. Ports define an interaction point between a class and its environment or a class and its contents. They allow you to group the required and provided interfaces into logical interactions that a component has with the outside world. Collaboration provides constructs for modeling roles played by connectors.

Comparing complex objects can be problematic, because often a whole object is considered to remain the same even though some of its parts might change. UML 2.0 supports:

- reference objects, which are equal if they have the same name regardless of content,
- value objects, which need to have the same content to be equal.

Although it is not strictly part of the complex object feature set, the feature template (parameterized class) is most useful where the parameterized class is complex.

4.3 Access control

UML permits a property to be designated read-only. It also allows classes to have public and private elements. It is proposed that the ODM (and OWL) omits access control features.

4.4 Keywords

UML has keywords which are used to extend the functionality of the basic diagrams. They also reduce the amount of symbols to remember by replacing them with standard arrows and boxes and attaching a <<keyword>> between guillements. A common feature that uses this is <<interfaces>>. It is proposed that the ODM (and OWL) omits this feature.

5 Converting UML to OWL

As we mentioned above our goal is to convert the UML model obtained from the requirements into a corresponding requirements ontology model. Such a requirements ontology model will compared with the domain ontology model and it will be investigated whether some new knowledge concerning the correctness, consistency, completness, and unambiguity could be made.

5.1 Existing Convertors

The UMLtoOWL tool by Gasevic [62] converts UML model description in extended Ontology UML Profile (OUP) using the XML Metadata Interchange (XMI) format to The Web Ontology Language (OWL) ontologies. The tool is implemented using eXtensible Stylesheet Language Transformation (XSLT). With UMLtoOWL we do not need to modify the existing UML tools. A UML tool (e.g. Poseidon) can export an XMI document that an XSLT processor (e.g. Xalan Java 2) can use as input. An OWL document is produced as output, and this format can be imported into a tool specialized for ontology development (e.g. Protégé) where it can be further refined [44], [36]. We think that it would be suitable to prefere this tool.

However, two other tools are available to translate from the XMI output from Poseidon into OWL in RDF/XML form. They are exff from Eurostep [55] and DUET [45] from AT&T. exff was a proof of concept and thus isn't fully functional. DUET preserves more of the semantics of the model through the translation, but that exploits more of OWL's expressivity. DUET also has a primitive user interface, so it may require some work to use.

6 Description Logics and Reasoning in Ontologies

6.1 Introduction

Description Logics (DL) is the most recent name for a family of knowledge representation formalisms that represent the knowledge of an application domain (”the world”) in a structured and formally well-understood way.

The name description logics is motivated by the fact that the important notions of the domain are described by concept descriptions. This means that expressions are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL.

On the other hand, DLs differs from their predecessors, such as semantic networks and frames, in that they are equipped with a formal, logic-based semantics, which can, e.g., be given by a translation into first-order predicate logic. Knowledge representation systems based on description logics (DL systems) provide their users with various inference capabilities that deduce implicit knowledge from the explicitly represented knowledge. Briefly, they:

- define the relevant concepts of the domain (its terminology),
- use these concepts to specify properties of objects and individuals occuring in the domain (the world description),
- are equipped with a formal, logic-based semantics,
- allow reasoning to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base,
- support inference patterns that occur in many applications of information processing systems: classification of concepts and individuals.

Classification of concepts determines subconcept / superconcept relationships (called subsumption relationships in DL) between the concepts of a given terminology, and thus allows one to structure the terminology in the form of a subsumption hierarchy. This hierarchy provides useful information on the connection between different concepts, and it can be used to speed up other inference services.

Classification of individuals (or objects) determines whether a given individual is always an instance of a certain concept (i.e. whether this instance relationship is implied by the description of the individual and the dfinition of the concept). It thus provides useful information of the properties of an individual. Moreover, instance relationship may trigger the application of rules that assert additional facts into the knowledge base.

The following three ideas have largely shaped the subsequent development of DLs:

- The basic syntactic building blocks are atomic concepts (unary predicates), atomic roles (binary predicates), and individuals (constants).

- The progressive power of the language is restricted in that it uses a rather small set of constructors for building complex concepts and roles.

- Implicit knowledge about concepts and individuals can be inferred automatically with the help of inference procedures. In particular, subsumption relationships between concepts and instance, relationships between individuals and concepts play an important role: unlike Is-A links in semantic networks, which are explicitly introduced by the user, subsumption relationships and instance relationships are inferred from the definition of the concepts and the properties of the individuals.

Most implemented DL systems provide for a rule language, which can seen as a very simple, but effective, application programming mechanism.

A DL system not only stores terminologies and assertions, but also offers services that reason about them. Typical reasoning tasks for a terminology are to determine whether a description is satisfiable (i.e non-contradictory), or whether one description is more general than another, that is, whether the first one subsumes the second.

Important problems are to find out:

- whether the assertions entail that a particular individual is an instance of a given concept description,

- whether its set of assertion is consistent, that is, whether it has a model.

Satisfiability check of descriptions and consistency checks of sets of assertions are useful to determine whether a knowledge base is meaningful at all.

With subsumption tests, one can organize the concepts of a terminology into a hierarchy according to their generality. A concept description can also be conceived as a query, describing a set of objects one is interesting in. Thus, with instance tests, one can retrieve the individuals that satisfy the query.

A restricted mechanism to add assertions are rules. Rules are an extension of the logical core formalism, which can still be interpreted logically.

For instance, the subsumption algorithm allows one to determine subconcept-superconcept relationships: C is subsumed by D iff all instances of C are also instances of D, i.e., the first description is always interpreted as a subset of the second description.

In order to ensure a reasonable and predictable behaviour of a DL system, the subsumption problem for the DL employed by the system should at least be decidable, and preferably of low complexity. Consequently, the expressive power of the DL in question must be restricted in an appropriate way. If the imposed restrictions are too severe, however, then the important notions of the application domain can no longer be expressed.

Investigating this trade-off between the expressivity of DLs and the complexity of their inference problems has been one of the most important issues in DL research but it is out of the scope of this report because we are interested in applications of existing methods and tools.

6.2 Basic definitions and foundations

The main expressive means of description logics are so-called concept descriptions, which describe sets of individuals or objects. Formally, concept descriptions are inductively defined with the help of a set of concept constructors, starting with a set N_C of concept names and a set of N_R role names. The available constructors determine the expressive power of the DL in question.

In the description logic $\mathcal{ALC}$, concept descriptions are formed using the constructors negation, conjunction, disjunction, value restriction, and existential restriction. The description logic $\mathcal{ALCQ}$ additionally provides us with (qualified) at-least and at-most number restrictions. Theoretical foundations can be found in Baader [7].

6.3 Using Description Logics for Data Modeling

Many applications require managing a symbolic model of an application world, which is updated or queried by users. Often, it is useful to think of various kinds of individuals (e.g. Calculus, Newton, Milner) related by relationships (e.g. taughtBy) and grouped into classes (e.g. Course, Teacher, Student). This intuition is shared by formalisms such as semantic data models, object-oriented databases, and semantic networks. Such formalisms support languages for declaring classes of individuals, using a syntax somewhat resembling the following example:

```
class Tropical Expedition is-subclass_of Expedition
  ...
  Participants : Alpinist
  ...
 with conditions
    (at-most 10 Participants)
    (all Participants Alpinist)
```

Example 1: Compositional concept in classic

Such a declaration is intended to express necessary conditions that must be met by each instance of the class.

For example, in the above case every instance of Tropical Expedition must also be an instance of the class Expedition, and the Participants attribute must relate to between 0 and 10 individuals, themselves instances of class Alpinist. Class definitions are used to detect errors, or as a template for data storage decisions, i.e. as a type declaration in standard programming languages.

Currently, there is another family of formalisms available - description logics (DLs).

6.3.1 Intesional and extensional approach

The fundamental observation underlying DLs is that there is a benefit to be gained if languages for talking about classes of individuals yield structured objects that can be reasoned with.

Example 1 a typical compositional description, expressed in the classic language. Its intended reading would be "Expeditions with at most 15 participants". In this description, Expedition and Alpinist are identifiers for concepts introduced elsewhere, while Participants is the name of a binary relation, intended to relate Expedition to Customers participating on them. There are several things, one can do with such a description, including:

- Reasoning about the relationship of one description to another, treating them as "intensional" objects. For this kind of reasoning we only need descriptions, i.e. only definitions. For example, the description in Example 1 is subsumed by (entails) the description

```
Def-1:    class Expedition
           with conditions
           (at-most 15 Participants)
```

since everything with at most 10 fillers for some role also has at most 15 fillers for it. On the other hand, the description

```
Def-2:    class Expedition
           with conditions
           (at-least 12 Participants)
```

can be inferred to be disjoint from the one in Example 1, because the required number of fillers are in conflict.

- Recognizing those individuals ("extensional" objects) that satisfy the description, based on what is currently known about them. For this kind of reasoning we need individuals (i.e. instances) including values of their atributes. For example, suppose Kilimanjaro is an individual object in the knowledge base, and it is known to be an instance of the concept Tropical Expedition, and in addition, the fillers for the Participants role for Kilimanjaro are individuals Custer and Milner, both of which are instances of Alpinist. Then Tropical Expedition is inferred to be an instance of the description in Example 1, since all necessary and sufficient conditions of that concept are satisfied if Expedition is defined according to the Def-1.

As a possible clarification of the issues involved, we provide an analogy for those familiar with logic programming. Since descriptions denote concepts or relationships, it is natural to take their analogues in logic to be ordinary unary or binary predicates. Consider the following knowledge base of Horn clauses:

$ParentOf(jane, fred).$
$Male(fred).$
$Child(x) : ParentOf(z, x).$
$Son(y) : Male(y), ParentOf(w, y).$

Normally, such a system is used to deduce new properties of individuals, e.g. whether the Son predicate "recognizes" the individual Fred.

On the other hand, we might want to reason entirely from intensional information - the rules - ignoring ground facts. For example, we might be interested in whether Child(x) is implied by ("subsumes") Son(x). Note that although we cannot express this question in Prolog, theoretically the answer would be "yes", because the last two clauses are in fact treated as the following definitions

$Child(x) \Leftrightarrow (\exists z)ParentOf(z; x)$
$Son(y) \Leftrightarrow (\exists w)Male(y) \wedge ParentOf(w; y)$

by the semantics of predicate completion in Prolog. However, if we took the rule for Son as its definition seriously, then asserting Son(fred) ought to allow us to deduce that Child(fred) — a deduction not made in current logic programming systems. It is such reasoning with definitions that is the trademark of description logics.

6.4 The Syntax and Semantics of DLs

All DLs have (at least) two sorts of terms:

- concepts (intuitively, denoting collections of individuals),
- roles (intuitively denoting relationships between individuals); because functional relationships occur very frequently, such roles are often distinguished, and will be called attributes.

Therefore the syntax of DLs consists of rules for creating composite terms from atomic/primitive symbols - identifiers of various sorts - and term constructors.
There is a fairly comprehensive list of domain-independent description constructors in [145] (including the exactly defined interpretation).
For concepts:

- top-concept
- nothing
- and[C,D]
- or[C,D]
- not[C]
- all[p,C]
- some[p,C]
- at-least[n,p]
- at-most[n,p]
- at-least-c[n,p,C]
- at-most-c[n,p,C]
- same-as[p,q]
- subset[p,q]
- not-same-as[p,q]
- fills[p,b]
- not-fills[p,b]
- one-of[b_1,...,b_m]

For roles:

- top-role,
- identity f(d; d)
- role-and[p,q]

- role-or[p,q]
- role-not[p]
- inverse[p]
- restrict[p,C]
- compose[p,q]
- product[C,D]
- trans[p]

These constructors were found empirically, in efforts to express the meaning of natural language sentences. In fact, one can view DLs as a logical notation where logical operators were chosen to facilitate the expression of frequently used conceptual structures, and related inferences.

6.4.1 The Logic of Descriptions

We have seen above that an interpretation associates an extent with every concept description, just like the interpretation of a unary predicate in First Order Predicate Calculus (FOPC). There are a number of natural questions that one normally asks about a descriptions C, D:

- Is D coherent/consistent?
  The answer is no if the denotation of D, $D^{\mathcal{T}}$, is empty for every possible relational structure $\mathcal{T}$.
- Does D subsume C?
  The answer is yes if the denotation of C is a subset of the denotation of D, $C^{\mathcal{T}} \subseteq D^{\mathcal{T}}$, for every possible relational structure $\mathcal{T}$.
- Are D and C mutually disjoint?
  The answer is yes if $C^{\mathcal{T}} \cap D^{\mathcal{T}} = 0$, for every possible relational structure $\mathcal{T}$.
- Are D and C equivalent?
  The answer is yes if $C^{\mathcal{T}} = D^{\mathcal{T}}$ for every possible relational structure $\mathcal{T}$.

The subsumption relationship, which corresponds to material implication between predicates and is symbolized by $\Rightarrow$, is usually considered the most basic one. This is because all DLs have concept constructors **and** and **nothing** (which denotes the inconsistent concept with empty extension), so that incoherence can be detected by asking the question “D $\Rightarrow$ nothing”, while disjointness is answered by “and(C; D) $\Rightarrow$ nothing”, and equivalence ($\equiv$) is mutual subsumption.

6.4.2 Reasoning with DLs

Although the original goal of DLs was to provide a convenient form for expressing the desired knowledge and inferences of some application, a highly influential paper [22], explored the idea that choosing a subset of concept constructors leads to description logics of more restricted expressiveness, but at the same time more efficient reasoning.

As a result, there is a large body of literature considering various combinations and variations of constructors for which reasoning is decidable, or even tractable. For purposes of this report we mention that the subset of concept constructors: and, all, at-most, at-least, same-as on attributes, fills, one-of with integers is polynomial [20].

6.4.3 Descriptions as constraints

We have already seen that it is useful to associate some necessary conditions that would have to hold of its individual instances with a primitive concept. It turns out that such a facility is very useful. Such a constraint might be stated using a constrain-type operator

```
constrain(<constrained-set>,<constraint-condition>)
```

where both arguments are descriptions.

6.4.4 DLs for stating rules

A more “active” system can be obtained through the addition of an operation such as

```
assert-rule(<lhs-descrn>,<rhs-descrn>)
```

This would have the effect that every time an individual is recognized as a left-hand-side, it would be added to the concept right-hand-side.

Such rules are less expressive than standard production rules because their antecedent is often only a single concept (rather than a relationship between individuals). But because of their treatment of incomplete information, rules based on DLs provide other advantages:

- classification applied to the antecedent (or even the consequent) of rules can be used to organize them into a hierarchy. This means that the system can help the programmer find closely related rules - a frequent cause of errors in rule-based programming.
- classification can also help implement the usual conflict-resolution strategy of “apply the most specific rule” by using the automatic classifier, rather than relying on the programmer to specify which rule is more specific.

Descriptions can be used in a natural way to specify a limited set of conditions and actions for a variety of rule languages, including integrity constraints, triggers, defaults, etc. In all such cases, subsumption can be used to organize large sets of such rules, and recognition helps in the firing process.

6.4.5 On the generality of the DL framework

It is important to point out the generality of the framework above. First, there is no reason to restrict the notion of “individual” to mean “object with intrinsic identity”. Therefore, it is entirely possible to consider mathematical entities (e.g. integers, n-tuples), programming language values (e.g. arrays, procedures), composite values (e.g. lists or trees of others kinds of individuals) as individuals, and have descriptions that denote sets of such individuals.

Second, there is a complete freedom in the choice of term constructors in the language syntax, and their intended interpretation.

For example, [13] introduces special concept constructors for describing classes of temporal intervals. Thus if we denote by time-interval:

```
after(2001) and duration-greater(5,year) and before(now)
```

refers to all time intervals beginning after 2001, of duration at least 5 years, which end before the reference time interval now. Such temporal concepts can then be used with constructors **sometime** and **alltime** to describe sets of individuals.

```
Profesor and sometime(time-interval,Student)
```

represents the set of individuals who are professors now and who were students for a period of at least 5 years between 2001 and now.

We have therefore two more observations:

- There is no “universal” set of term constructors. The term constructors used in a DL may be domain or even application specific.
- The denotations of concept descriptions do not have to be atomic individuals, but could have internal (mathematical) structure.

6.4.6 Complexity vs. expressiveness

We have already mentioned the strong interest concerning the decidability and complexity of reasoning with various DLs. There are some aspects to be mentioned:

- Limited languages - Some authors have argued that DL-based systems need to respond in polynomial time if they are to be useful as “servers” to other problem solvers.
- Complete reasoners for intractable languages - Some researchers think that as long as the logic is decidable, it is reasonable to deliver to the users a system that reasons correctly with it. The main obstacle of this approach is to make the performance of the system predictable, so that users are aware of the forms of knowledge which can cause exponential explosion in the time or space used by the system.

We remark that certain worst-case complexity results - such as the result that just by allowing definitions can lead to an exponential blow-up during processing [101] - are not considered to be a problem, because the examples are pathological and do not arise in practice. The conclusions in this section are that the conflicting between expressive languages and complexity of reasoning, although very real, need not to be paralyzing. There is wide variety of approaches to the problem, with the “predictability” of the inferences and their timing being of concern to users.

Description languages provide a variety of constructors for building terms that can be used to express knowledge about the world. They have found applications in a variety of areas such as data management, linguistics, and knowledge-based software engineering [43].

7 Existing Reasoners in Ontologies

7.1 Using RacerPro for Reasoning in Ontology

As we mentioned above the advantage of using ontology for our purpose is in applying reasoning. It means that we do not only need an ontology editor like Protégé, but also a reasoner. The reasoner RacerPro [117] (stands for Renamed ABox and Concept Expression Reasoner Professional) is one of best in this field. This reasoner is recommended as a reasoning engine for ontology editors such as Protégé. RacerPro can process OWL Lite as well as OWL DL documents. However, some restrictions apply, however. OWL DL documents are processed with approximations for nominals in class expressions and user-defined XML datatypes are not yet supported.

The following services are provided for OWL ontologies:

- Check the consistency of an OWL ontology and a set of data descriptions.
- Find implicit subclass relationships induced by the declaration in the ontology.
- Find synonyms for resources (either classes or instance names).
- Since extensional information from OWL documents (OWL instances and their interrelationships) needs to be queried for client applications, an OWL-QL query processing system is available as an open-source project for RacerPro.
- HTTP client for retrieving imported resources from the web. Multiple resources can be imported into one ontology.
- Incremental query answering for information retrieval tasks (retrieve the next n results of a query). In addition, RacerPro supports the adaptive use of computational resources: Answers which require few computational resources are delivered first, and user applications can decide whether computing all answers is worth the effort.

Details about using RacerPro are given in [117].

7.2 Using Jess for Reasoning in Ontology

The necessity to combine Protégé with a reasoner occurs very often. An interesting solution is offered by the system Jess [82] from Scandia and especially JessTab [83] from University of Linköping. JessTab is a plug-in, which integrates Protégé with Jess, a fast rule engine. JessTab lets us build knowledge bases in Protégé that work with Jess programs and rule bases, i.e. JessTab maps instances of the Protégé knowledge base to Jess facts. JessTab can also propagate modifications automatically to mapped Protégé instances on Jess. JessTab provides functions for managing Protégé knowledge bases. Because Protégé and Jess are implemented in Java, we can run them together in a single Java virtual machine. This approach lets us use Jess as an

interactive tool for manipulating Protégé ontologies and knowledge bases. Furthermore, we can propagate changes in the Protégé to Jess [54].

We suppose that JessTab combined as a plug-in with Protégé will be used in implementation of the proposed project.

8 The Proposed Concept

8.1 The existing tool TESSI Architecture and Dataflow

We argue that there is a gap between the requirements definition in a natural language and the requirements specification in some semi-formal graphical representation. The analyst's and the user's understanding of the problem are usually more or less different when the project starts. The first possible point of time when the user can validate the analyst's understanding of the problem is when a prototype starts to be used and tested.

In our approach [93], [94] that will be developed further in this report, we offer a textual refinement of the requirements definition which can be called requirements description. Working with it, the analyst is forced by our supporting tool to complete and explain requirements and to specify the roles of words in the text in the sense of object-oriented analysis. During this process, a UML model will be built with our tool driven by the analyst's decisions.

This model will be used:

- for checking ambiguity of the described requirements,
- for checking for inconsistency and completness using the domain knowledge described in domain ontology,
- for the synthesis of a text that describes the analyst's understanding of the problem, i.e. a new, model-derived requirements description will automatically be generated. Now, the user has a good chance to read it, understand it and validate it. His/her clarifying comments will be used by the analyst for a new version of the requirements description.

The process repeats until there is a consensus between the analyst and the user. This does not mean that the requirements description is perfect, but some mistakes and misunderstandings are removed.

We argue that the textual requirements description and its preprocessing by our tool will positively impact the quality and the costs of the developed software systems because it inserts additional feedbacks into the development process.

As we have already mentioned we have been working on a tool that helps analysts align their views of the problem space with their users views for many years [93], [94]. Our tool should support all phases of requirements engineering from the begin when an analyst takes notes during user interviews.

The steps of the proposed requirements processing that uses ontologies are the following:

- building a domain ontology in OWL using Protégé,
- checking a domain ontology for corretness and consistency using RacerPro and Jess or JessTab,
- building a UML model of requirements from a textual description of requirements,

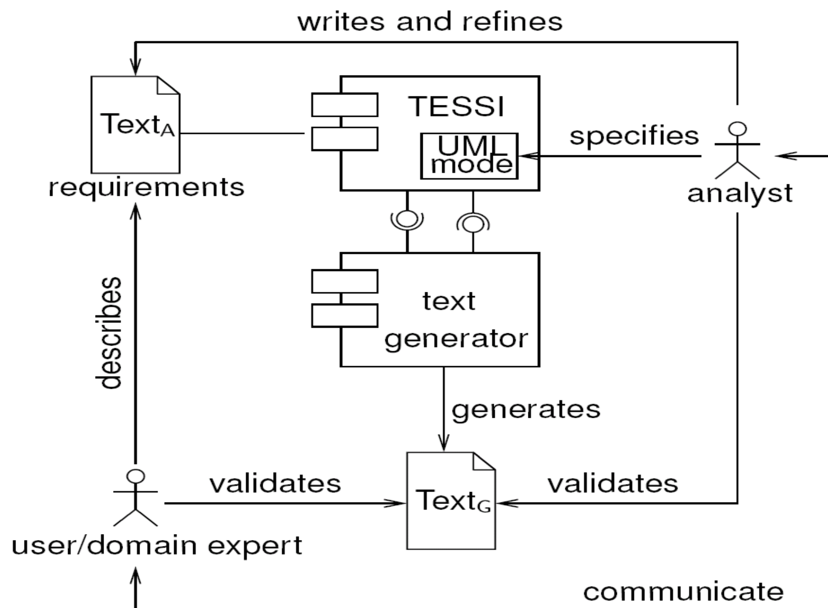

Figure 8.1: Architecture and dataflow of the existing tool TESSI

- conversion of requirements described as a UML model to a requirements ontology in OWL and its limits,
- checking requirements tranformed into the requirements ontology for mutually correctness and consistency checking and for checking with respect to the domain ontology,
- identifying correctness and consistency problems,
- finding the corresponding parts in the former textual description of requirements and correcting them,
- building a new UML model based on corrected textual description of requirements,
- after iterations when no ontology conflicts will be found a new textual description of requirements will be automatically generated that corresponds to the last UML model,
- before the UML model will be used for design and implementation the customer and the analyst will read the generated textual description of requirements and look for missing features or misunderstandings,
- problems found can start the next iteration from the very beginning,
- after no problems have been found the UML model in form of a XMI-file will be sent to Rational Modeler for further processing.

8.2 The proposed extensions of the tool TESSI

In the proposed extension of our tool new components will be included:

- OWL-component
- CONVERT-component

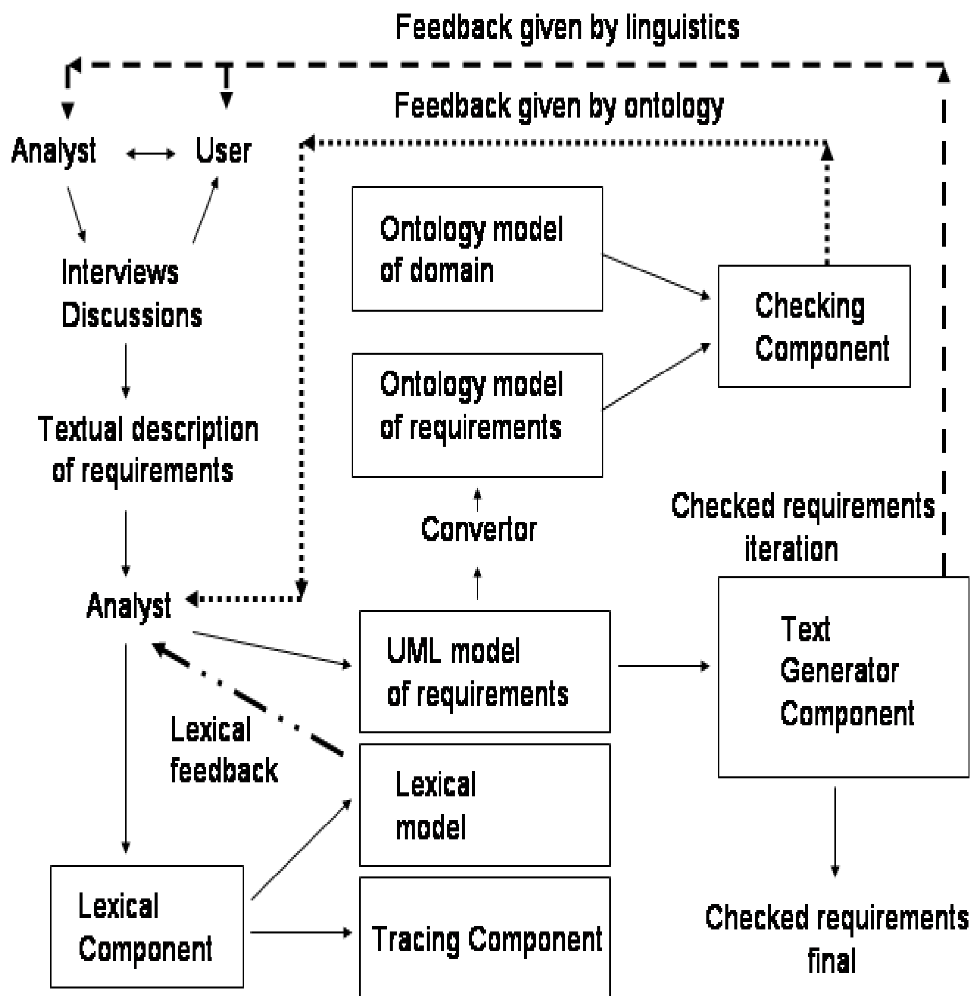

Figure 8.2: Architecture and dataflow of the proposed tool TESSI

Using experiences given in [96], we describe the ontology in Protégé and apply ontology reasoning (e.g. the inference engine in RacerPro [23], [72] - for checking classes) first for domain ontology checking, then for requirements ontology checking, and last for checking whether the requirements ontology subsumes the domain ontology.

We propose to use rules based on SWRL language to express domain rules, and the Jess [84] reason engine to check rules. As mentioned above we will experiment with JessTab. The other possibility is to use Tab Widget [135] as a plug-in component for SWRL into Protégé that allows mapping SWRL rules into Jess rules and OWL instances into Jess facts. The tool TabWidget builds a bridge betweeen RacerPro and Jess.

We will check the suitability and performance of systems mentioned above and find the most simple solution. At the very moment we mean that Protégé and JessTab (as its plug-in) can fulfill all tasks.

The process of requirements checking can run as follows:

- Ontology part:
  - Domain ontology description based on OWL and SWRL based is constructed by domain experts at first.

- The domain ontology description in OWL (Protégé) will be transformed into the concepts set and roles set of RacerPro and the SWRL rules will be transformed into Jess rules by the tool TabWidget.
- RacerPro inference engine will be used to do ontology checking, which includes satisfiability, subsumption, equivalence and disjointness checking of concepts in ontology.
- The instances of concepts and roles will be sent from RacerPro to Jess, so Jess can start the inference machine having rules from SWRL and facts from Racer. The results of inference will be sent from Jess to RacerPro. It is possible that the whole inference can be made by JessTab. I this case, RacerPro will not be needed.
- The checking results are returned to domain experts so that they can revise the ontology.

• Requirements part:
  - The analysts can establish application requirements in UML guided by the ontology.
  - Analyst converts the requirements in UML into OWL. There are some tools available e.g. Gasevic [62].
  - The requirements converted via UML into ontology in OWL will run through the same process as the domain ontology.
  - According to the results of checking, analysts modify the requirements to make them comply mutually and with the domain ontology.
  - Domain ontology and domain rules will be composed together with requirements ontology and requirements rules (constraints). This new composed ontology will be checked again using the same procedure as before.
  - In conclusion, the RacerPro disposes integrity rules of the ontology model and Jess deal with derivation rules of the requirements model so that the acquired requirements will comply with, both business needs and domain knowledge.

One of the problems that may occur is that the restriction rules of requirements (called constraints) are described in OCL (Object Constraint Language) which is stronger than SWRL. As we will describe below description logics have different expresiveness and the reason is that the computational complexity of the reasoning, i.e. of the decision whether the system is correct and consistent, may explode and we never obtain the result guarenteed if the expressivenes of the used description logic is too high.

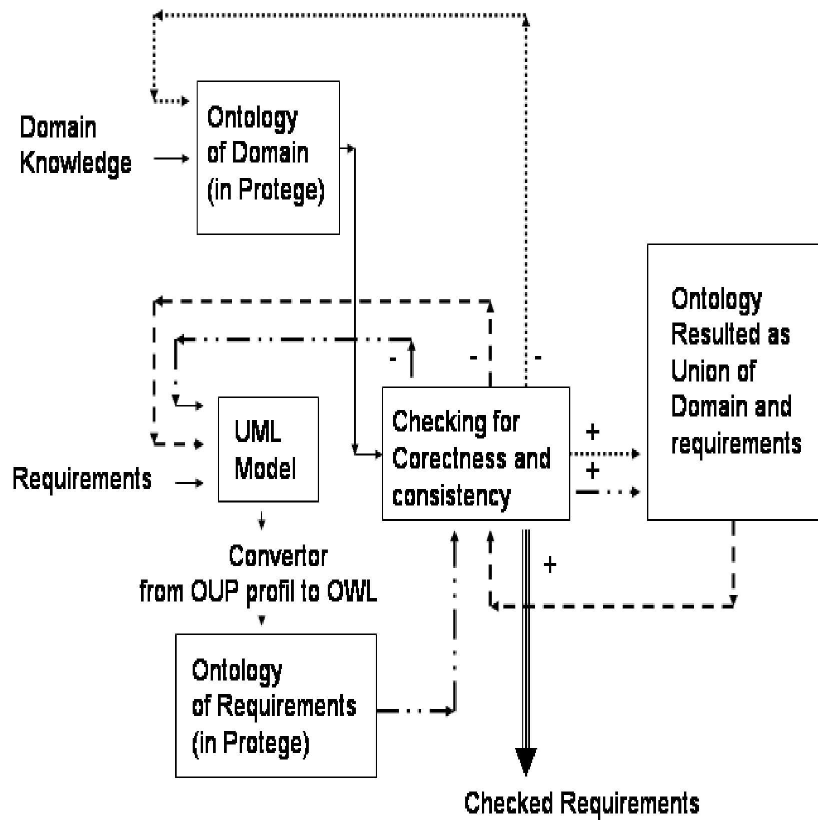

Figure 8.3: Checking requirements process

9 Requirements Engineering

In this chapter we describe briefly the parts of requirements engineering focused on checking correctness, consistency, completness, and unambiguity that concern the problem discussed in this report.

Requirements engineering identifies the purpose of a software system, and documents it in a form that is suitable to analysis, communication and subsequent implementation [111].

If a software system shall be build it has to be described in some way before the design and implementation process will be started. Typically, these descriptions, requirement documents, are far from representing the real business logic [29]. Instead, we have a set of statements that is:

- incomplete (forgotten features),
- inconsistent (contains contradictions),
- ambiguous (more possible interpretations).

As software requirements quality we may denote the degree in which the requirements are incomplete, inconsistent (contradictory), and ambiguous. The importance of software requirements quality has long been recognized, and it has been estimated that an error that is not identified and corrected during the requirements phase can cost as much as two hundred times more to correct in subsequent phases [15]. That is why this topic plays an important role in software development and why there have been hunderds of publications trying to solve this problem in the last twenty years.

9.1 Domain Knowledge and Project Requirements

Before a software system can be implemented, the customers' needs must be found, described, and finally analyzed as requirements.

We distinguish the following kinds of requirements [38]:

- Project (application) requirements - requirements that reflect the user needs
  - Functional requirements - statements of services that the system should provide, how the system should react to particular inputs and how the system should behave in particular situations
  - Non-functional requirements - requirements on the services or functions offered by the system such as performance, security, constraints on the development process, using standards, etc.
- Domain requirements - requirements that come from the domain of the system and reflect the characteristics of that domain. They describe all objects, their properties, relations, possible states, and constraint. They may be functional or non-functional.

We can identify the following phases of the requirements processing:

- requirements elicitation
- requirements checking
- requirements analysis

Below we will describe the phases in more details.

A very natural way is to say that project requirements should be a subset of domain requirements because the project requirements cannot contain features that are not in domain requirements. The subset is given by the set of services, e.g. queries, that the user needs. Speaking about requirements usually means speaking about project requirements. Domain requirements will be often denoted as domain knowledge. It is a simplification because the needs of user concern also properties of output data presentation (for example), which are not directly part of the domain knowledge. Such properties are sure part of the universal domain of all possible output data presentation. However, focusing on functional requirements can bring important results.

Confronting project requirements and domain requirements in the sense described above we can find that in project requirements:

- there are different names and symbols used as in domain requirements,
- there are some names and symbols used in another meaning than in domain requirements,
- there is something that cannot be found in domain requirements,
- there is something that is in contradiction with domain requirements.

The results found can overlap and the situation must be analysed.

In an ideal case we could hold the domain requirements as a reference requirements and declare that they are complete, do not contain any mistakes and contradictions. This is never guaranteed, of course, but usually the domain requirements are known better than the project requirements. Because of that, the mistakes are to be searched first in the project requirements.

There are in principle three methods, how to elicitate and specify domain requirements [95]:

- After discussions with domain experts and users, analysts describe the domain requirements in natural language. The main drawback of this method is that there are gaps in the knowledge between domain users and requirements engineers as mentioned above. Domain users cannot communicate the requirements in the terminology that analysts want. This will be the main source of generating incomplete, inconsistent and ambiguous requirements.
- After discussions with domain experts and users, analysts describe all scenarios that users want to use. Users can take part in the elicitation conveniently, and the requirements are the true reflection of the real system. The problem is to be sure that the found set of scenarios represents the complete requirements of the system [86]. Moreover, different scenarios represent usually also different views that overlap and are the source of requirements inconsistency.
- The youngest method is based on domain knowledge, uses ontologies and their possibility of reasoning. This field has been researched intensely. We will use it here.

Our approach combines the advantages of all three methods, i.e. we use textual description of the requirements containing use-cases, then we build the corresponding UML model and Lexical model and as the last phase of one cycle we convert the UML model into an OWL DL ontology and start reasoning procedures searching for aspects hurting correctness and consistency.

Using the Lexical model we can eliminate some natural language sources of ambiguity, inconsistency, and incompleteness bevor the modeling starts.

9.2 Requirements Elicitation

To find the needs of users, analysts:

- do interviews,
- do observations,
- study the domain,
- study existing solutions and similar systems.

Eliciting knowledge via communication is necessarily fraught with difficulty. However, it is more or less this is the only way how an analyst can learn about the user, his needs and behaviors. This knowledge is what permits the analyst to determine system requirements and specifications. Knowing the user, the analyst can know what will work for the user. The communication (exactly said, the knowledge resulting of the communication) is that, upon which the success of the project rests.

Because of mistakes during requirements elicitation many projects go over schedule, go over budget, and are replete with errors which require more time and more money to correct. Often users feel dissatisfied with a product that in no way resembles what they had expected or that does not accurately mirror their needs.

The communication language between the analyst and the user is the natural language, so part of requirements engineering focus on an analysis of words, phrases, sentences - arguing that this is where the solution resides. The problem space of the form and structure of natural language expressions will be used to classify context and knowledge contained in textual description of requirements.

The analyst has to map informal understanding obtained by an interview and described as a text into a formal information structure like a UML model. It is a collaborative act where both - interviewer and user - are engaged in creating the meaning of the questions and answers as they negotiate understanding through language. The meaning that is constructed is a representation of the problem domain.

This process brings the following problems:

- In order to successfully acquire user requirements, an analyst should understand the users activities, motivations and negotiate issues of organizational context which have some influence on the development process. This is difficult.
- Analysts and users typically do not share common experiences, vocabulary, and background. This complicates communication between the two.

- Interview participants have a private world that must be transformed into public languages. Each participant has his own level of knowledge and expertise, skills, and requisite vocabulary. What one thing means to one person, may have a completely different meaning to another.

- When these concepts of an analyst und user are represented formally in functional specifications documents, they tend to shift in meaning when passed between different semantic communities [118].

- Users do not have the same level of technical knowledge as the analysts.

- Analysts do not have domain or business knowledge and consequently tend to misunderstand or ignore some requirements and their context.

The communication problems are introduced and analysed in details in many publications [73], [85], [31].

The process of requirements specification, starting after (sometimes also during) the requirements elicitation, usually run as follows.

- Before the modeling phase of the software development can start, requirements are acquired and collected. Using natural language is necessary because a customer would not sign a contract that contains a requirements definition written in some formal notation (e.g. in Z-notation).

- The interviews and studies result in text documents that describe requirements.

- After discussions about the contents of these documents, a requirements specification must be written that describes the functionality and the constraints of the new system in a more detailed way. The non-functional requirements are usually appended later, after the functionality has been specified.

- Requirements specification is usually written as a combination of text and some semi-formal graphical representation given by the used tool. Since software engineers are not specialists in the problem domain, their understanding of the problem is immensely difficult, especially if routine experience cannot be used.

It is a known fact [29] that projects completed by the largest software companies implement only about 42 % of the originally proposed features and functions.

9.3 Requirements Analysis

Early requirements analysis is one of the most important and difficult phases of the software development process. It is the phase where the requirements engineer is concerned with understanding the organizational context for an information system to be developed, and the goals and social dependencies of its stakeholders. This phase demands critical interactions with users and other stakeholders. A misunderstanding at this point may lead to expensive errors during later development stages [66].

The requirements elicitation and the requirements analysis overlap in many cases and the processing is iterative because the gained knowledge is not sorted and an analyst will not get it in some predefined order.

Usually, we cannot separate clearly the phase of requirements elicitation and requirements analysis because during the requirements elicitation some analytical decisions are made and during the requirements analysis it may happen that additional requirements have to be acquired.

As a results of the requirements analysis a model will be constructed. In this report we suppose that it is a UML model. The transformation of the requirements specification in a textual form (it will be denoted as requirements description here) into a UML model is a creative activity and depends on the experiences of the analysts.

In our tool we support this activity by the method of grammatical inspection [123], [94] used for building the UML model from textual description of requirements.

10 Related Work to Checking Requirements Specification

In this chapter we discuss works and research papers that are related to the topic of checking a requirements specification. It is known that writing requirements specification without any checking brings problems.

It is known that the quality of the software depends on the requirements upon which the system has been built. The quality can be measured, e.g. as the number of failures and problems reported by users after the software system has been delivered. Comparing the costs to correct defects in software discovered at various phases of the lifecycle we can say that the later in the software lifecycle a defect is discovered the more expensive it is to rectify.

Some studies have shown that correcting software defects can require nearly two hundred times more effort if the correction is implemented in the maintenance phase instead of in the requirements specification phase of a software lifecycle [39].

There are many possibilities to check requirements. They will be described in this chapter.

10.1 Checking Unambiguity of Requirements Specification

Ambiguity of natural language means that different readers of the requirements specification may understand different things. If the implementors' understanding of the document differs from that of the customer or users, then the customer and the users are likely not to be satisfied with the implementation produced by the implementors. Ambiguity that remains in a final natural language requirements description documents is a major problem in requirements specification.

Many times, ambiguity is not noticed by anyone looking at the requirements document. Very often, the reader disambiguates the document during the first reading subconsciously to the first interpretation he finds and thinks that this first interpretation is the only interpretation.

In industrial requirements engineering, natural language is the most frequently used representation to state the requirements that are to be met by information technology products or services. Natural language is universal, flexible, and wide-spread, but unfortunately also inherently ambiguous. Even worse, often neither customers nor software developers recognize an ambiguity, and each derives an interpretation that differs from that of others without noticing this difference.

Consequently, software developers design and implement a system that does not behave as intended by the customers. Semi-formal and formal representation techniques, such as UML, have been proposed to overcome the weaknesses of natural language. However, they only shift the problem. An ambiguity simply becomes an unambiguously wrong specification statement, which must be detected by the customers in reviews or simulations of the specification.

The problem of detecting ambiguities in natural language requirements has been addressed in [12]. We distinguish:

- linguistic ambiguities - are the commonly known ambiguities of natural language, such as ambiguous pronoun references.
- requirements specific ambiguities arise from the requirements context, which comprises the application, system, and development domain.

10.1.1 Linguistic Ambiguities

The thesis [90] presents the ADTD (ambiguity detection technique for the domain) approach, which allows developing techniques for detecting ambiguities in natural language requirements on the basis of existing and industrially proven techniques, namely checklists, scenario-based reading, and agendas. A technique for detecting ambiguities must be tailored to a particular requirements context to be effective, because requirements-specific ambiguities depend heavily on the context in which an requirements specification takes place. For this purpose, the ADTD approach provides heuristics to investigate metamodels, which capture information about an requirements context. These heuristics help identify the requirements-specific types of ambiguities that are typical for the particular requirements context.

A set of ambiguity detection techniques for the domain of embedded systems has been developed using the ADTD approach to show its feasibility. To quantify the benefits of the ADTD approach, the resulting techniques were empirically validated in five experiments. The goals were:

- to validate that the investigation of metamodels pays off in terms of higher efficiency of the resulting technique,
- to show that our ambiguity detection techniques are superior to existing ones.

The handbook [11] and the work [12] - written together with a lawyer - stems from the observation that software requirements specifications and legal contracts are similar in several key aspects. Particulary when these are written, as they usually are, in natural language, ambiguity is a major cause of their not specifying what they should. Simple misuse of the language in which the document is written is a significant source of these ambiguities.

The common aspects of software requirements specifications and legal contracts are:

- each is usually written in natural language,
- each must anticipate all possible contingencies,
- each must be correct, consistent, complete, and unambiguous.

The handbook [11] details the problems arising from common difficulties with:

- “all”, “each”, and “every” to denote sets,
- positioning of “only”, “also”, “even”,
- precedences of “and” and “or”,
- “a”, “all”, “any”, “each”, “one”, “some”, and “the” used as quantifiers,
- “or” and “and/or”,
- “that” vs. “which”,

- multiple adjectives.

The handbook is suggested as a guide, both for writing better requirements or contracts and for inspecting them for potential ambiguities. An open problem remains the effectiveness of this handbook, both as a guide for writing specifications and as a guide for checking specifications.

10.1.2 Requirements-specific Ambiguities

Requirements specific ambiguities arise from expressions that can have a different interpretation for different participants of the requirement development process. This interpretation depends on the problem context and on the implicite knowledge of the participant.
In Parnas [114], they give an example of such a requirements specific ambiguity in a requirement about a continually varying water level in a tank:

- Shut off the pumps if the water level remains above 100 meters for more than 4 seconds.

They claim that this type of ambiguity is very common in informal requirements documents. One can find four interpretations:

1. Shut off the pumps if the mean water level over the past 4 seconds was above 100 meters.
2. Shut off the pumps if the median water level over the past 4 seconds was above 100 meters.
3. Shut off the pumps if the root mean square water level over the past 4 seconds was above 100 meters.
4. Shut off the pumps if the minimum water level over the past 4 seconds was above 100 meters.

However, the software engineers did not notice this ambiguity and quietly assumed the fourth interpretation. Unfortunately, under this interpretation, with sizable rapid waves in the tank, the water level can be dangerously high without triggering the shut off. In general, the interpretation of the ambiguity is very much a function of the readers background. For example, in many other engineering areas, the standard interpretation would be the third.

10.1.3 An Approach based on Neuro-Lingustic Programming

There is an approach described in Goetz [64] which was transferred from the discipline of psychotherapy to the field of requirements engineering as a psycho-therapeutic approach known as Neuro-Lingustic Programming (NLP) Bandler [8]. A set of rules was formed to assist the analysis and quality assurance of customer requirements represented as a text. Psychologist John Grinder and computer scientist Richard Bandler developed a therapeutic method based on the necessity of understanding the personal truth of the client. The therapist has to find out what the client really thinks (consciously, unconsciously or even under-consciously), when he says something. The method helps finding ambiguous, incomplete and inconsistent statements in specifications in a systematic way.
If the requirements are already defined, they are checked with the set of rules. Distortions, generalizations and deletions are considered closely.
Of the 25 rules, 17 are derived from the linguistic transformational effects used in NLP. The rest is derived from every-day experiences with real specifications and their defects and they can be found in requirements engineering books. Two basic principles rule the humans perception and his communication:

- focusing - during perception we are focusing and some parts of the reality never will be a part of our knowledge or will get quite abstracted.
- simplification - during communication, which is a kind of presentation, we are simplifying. As authors we assume that the reader has certain previous knowledge. Otherwise it would not be possible to communicate efficiently.

These two principles provide a transformation that is a source of problems not only in requirements specification but in the whole field of perception and communication.

Linguistic transformational effects that are most common among requirements in text:

- Deletions - Deletion is the linguistic counterpart of perceptional focus. It reduces statements, so that we can cope with them, whereas focusing is the process to concentrate our perception only on certain dimensions of experience. This reduction is quite helpful in normal communication but really hinders requirements definition, where important information is lost due to deletion. Under-specified process words are a typical case.
  - Presuppositions
  - Incomplete comparisons
  - Modal operators of imperative
  - Modal operators of possibility
  - Under-specified process words
- Generalizations - To abstract from concrete experiences is useful and vital for humans. By doing so we can transfer an experience we made to similar situations. It is important to wisely choose the set of situations we should transfer an experience to, though. In systems engineering over-generalized requirements ensure that special cases and exceptions will be omitted. As soon as the missing special case occurs during the life of the developed system the system doesnot behave as intended but as specified. A very easy to find indication of an over-generalization is universal quantors.
  - Universal quantors
  - Incomplete conditions
  - Nouns without reference
- Distortions - By distorting perceptions we prevent that our knowledge has to be reorganized constantly. Details are changed if necessary to fit in the picture we have already drawn from a given situation (structures of functional verbs). Similarly distortion takes place in utterances, the so called nominalizations are of particular interest concerning requirements documents.

Under-specified process words are a very important part of the linguistic Deletion transformation. Process words are the parts of a sentence that describe the action taking place. To be complete, there must be certain arguments or noun phrases around the process word.

Here is an example:

```
Rule no. 3
Uncover under-specified process words!
is applied to
The system shall report data losses.
```

The process word report is completely specified only if the following questions are answered:

- Who reports?
- How is reported?
- What is reported?
- Where and to whom is reported?
- When will the report take place?
- For how long shall the report be presented?
- ... etc.

Universal quantors are a very important part of the linguistic Generalization transformation. They state frequencies, e.g. never, always, none, all, any. The danger of using universal quantors is the fact, that the specified behavior of the system might not be actually true for all elements of the mentioned group of elements.

```
Apply rule no. 9
Find and question all universal quantors!
to
Each report shall be labeled with a time stamp.
```

Because of the signal word ”each” you should ask: really each report? Or are there cases where the time stamp is not necessary? Maybe only reports that are relevant for a specific subset of users (sales people?) shall be labeled.

There are many over-generalizations in specifications especially in description of systems correct behavior in exceptional situations. Users tend to dismiss exceptions in discussions because they are so hard to elicit and users are used to improvize in such cases.

The nominalization can change the meaning of a statement or important information is lost that. A nominalization has been applied to a process if a process word (verb or predicate) is transformed to an event word (noun) within a sentence.

```
The corresponding rule no. 17 is
Question all nominalizations!

In case of a system break-down an automatic re-start
shall take place.

The processes behind the nouns break-down and restart
are actually:

The system breaks down and the system restarts.
```

The second sentence alone can be further scrutinized by asking:

- What does restart?
- Which data shall be applied to the restart?

- How is the re-start conducted?
- Who initiates the re-start?
- When does it end?
- What happens during the re-start (exceptions, errors, etc.)?

The requirement shall therefore be supported by a clear definition of ”break-down”'. A definition of an automatic restart shall be given by various requirements concerning the re-start itself, e.g. that automatic restarts shall be logged.

In principle, one can use nominalizations for an otherwise hard to describe process as long as the process is very clear or defined unequivocally. Nominalizations especially occur in domains that have developed an esoteric language. Check some words that are specific to the domain. In most cases they are nominalizations behind which a lot of knowledge is hidden. By the way, computers can assist in finding nominalizations in lengthy documents.

In projects where completeness of the specification is crucial, the application of the set of rules resulted in more and more precise requirements. Typically the specifications size increased by 100 - 300 %.

Experience shows that the full application of the set results in requirements that are in fact no longer plain prose but semi-formal language. Sentences tend do be longer and the sentence structure tends to be more complicated, but with a limited grammar. This makes requirements less readable. So the analyst has to carefully judge this risk against the advantages.

It has been found that the set of rules should be applied as a supplement to techniques like use-case analysis or object-oriented modeling. Above all they deal with the context of the requirements, whereas the Set of rules mainly deals with single requirements. Other viewpoints find other defects.

```
The Set of rules:

Rule 1 - Phrase each requirement in active voice.
         Advantage: the actor has to be stated.
         The item list shall be printed.  to
         The system shall print the item list.
Rule 2 - Use main verbs for expressing processes.
         Adjectives or complex phrases camouflage
         the process. Verbs require more phrases
         to be complete.
         A decision has to be made between the
         various loan items. to
         The system shall differentiate between
         the various loan items.
Rule 3 - Uncover under-specified process words.
         Process words usually need a lot of
         arguments to be complete.
         If missing but important they
         should be requested.
Rule 4 - Complete incomplete comparisons (if any).
         Each comparison needs a reference point.
         Other customers shall only see the items
```

in stock. to
Other compared to what?
What kinds of customers do exist?

Rule 5 - Clarify the modal operators of possibility
(if any).
Can, Must, is allowed to, have to,
Why is something possible or impossible?
The parts of the Customer Component mustnt
forward data to the Main System. to
The Security Module shall prevent the Customer
Component from forwarding data to
the Main System.

Rule 6 - Clarify the modal operators of imperative
(if any).
Have to, should, shall, must, necessary.
Are there any exeptions?
The system shall provide the user a
back up procedure.
Always?
What if the procedure cannot be conducted
due to technical problems?

Rule 7 - Specify the implicit assumptions
(presuppositions) within the requirement.
Phrase them explicitly.
If the individual loan limit is reached
the system shall send a message to the
borrower. to
There is a loan limit, which one.
Loan limits can be exceeded, how?
Loan limits can be individual, concerning what?
There is a role borrower, are there other roles?

Rule 8a - Write an extra requirement if the relation
between the different objects of the
requirement (if any) IS important.
The system shall provide the ability to read back
the statistics on past lending processes. to
The system shall record the number of
loan items per user per year, sorted by
date of return, and the mean time between
lending and return.

Rule 8b - Write and use a definition if the relation
between the different objects of the
requirement (if any) is NOT important.
The system shall provide the ability to
read back the past lending statistics.

Rule 9 - Find and question the universal quantors
of the requirement (if any).
Always, every, never, no, only,
Ask for exceptions.

Each message shall be labeled by a time code. to
Messages of type Y, X and Z shall be labeled
by a time code.

Rule 10 – Define what the quantitative information
within a requirement stands for.
Any, each, every, either, neither,
get mixed up occasionally.
The system shall provide every user the
ability to read back each lending statistic.
to
provide users the all lending statistics.

Rule 11 – Complete incomplete conditions (if any).
If then. Else?
If the automatic data transfer was
malfunctioning the system shall provide the
ability to manually enter the customer data.
What if it worked properly? Can the user
still enter the data manually?

Rule 12 – Question nouns without references (if any).
Data shall be presented graphically to
the user.
to
The system shall present the lending
statistic to A-users graphically.

Rule 13 – Use singular nouns only (if applicable).
Do so if you dont want to refer to a set.
The names of all belated borrowers shall
be displayed in a list.

Rule 14 – Specify definite and indefinite articles
(if any).
A loan item has to have a ID number. to
Every loan item has to have exactly one
ID number.

Rule 15 – In definitions of nouns, use only
indefinite articles before the noun.
Otherwise the numbers may be confused.
The loan item is an item which
to
A loan item is

Rule 16 – Within a requirement, use a definite article
before a defined noun.
Again, to clarify the number.
The system shall find a loan item with
the given ID number.
to
The system shall find the quoted loan
item with the given ID number.

Rule 17 – Question the nominalizations of the
requirement (if any).

Isnt the noun a verb actually?
A message shall be sent on the return.
to
What initiates the process of returning?
What does it do? When will it be finished?

Rule 18 – Replace complex structures of functional verbs with simple main verbs.
To improve readability.
to be in use
to
to use,
to bring to an end
to
to end

Rule 19 – Avoid expressing something twice.
Again, to improve readability.
a true fact, yellow in color

Rule 20 – Delete or replace flowery phrases.
due to the fact that to because,
along the lines of to like

Rule 21 – Make comments from subordinate clauses with rationales, intentions or consequences.
The system shall sort the list alphabetically to improve its readability.
to
The system shall sort the list alphabetically.
Comment:
The users think alphabetical sorting is more readable.

Rule 22 – Start clauses that have a temporal relation to the main clause with a when or once.
Otherwise it could be interpreted as logical relation.
If the list is compiled, the system shall print it.
to
Once the list is compiled, the system shall print it.

Rule 23 – Define nouns like noun = verb + object(s) + additional phrases.
A borrower lends loan items from the library.

Rule 24 – Define adjectives like adjective + noun + is + noun + additional phrases.
A loan item is a removable object of the library which can be lent to a borrower.

Rule 25 – Define verbs like infinitive verb + is the process of + additional phrases.
Lending is the process of temporarily removing a loan item from the library.

10.2 Checking Inconsistency of Requirements Specification

The requirements elicitation and specification involves the collaboration of many participants. Their partial models describe the system from different angles and in different levels of abstraction, granularity and formality. They may also be constructed using different notations. This results in many views on the domain parts and many partial models that can overlap and because of the overlapping they can be inconsistent since they describe the system from different perspectives and reflect the different views of the stakeholders (Fig. 10.1).

Inconsistencies arise because:

- The submodels overlap and their overlapping parts are inconsistent, i.e. they incorporate elements which refer to common aspects of the system under development, and make assertions about these aspects which are not jointly satisfiable.
- A software model is inconsistent according to the domain, i.e there is a consistency rule in the domain and it can be shown that the consistency rule (usually some general knowledge about the domain) is not satisfied by the model.

In [131] only the inconsistence between overlapping models has been discussed.

Example:
In Fig. 10.1 the problem is that two submodels are using their own coordinates (from me to the left) instead of using objective coordinates (from the North to the South). We can imagine an assertion as a part of the domain that says ”the object X is a real estate” and a constraint rule ”real estates cannot move”. We found an inconsistence in submodels but we could solve it. Then we found an inconsistence between the model and the domain, which cannot be solved.

The model is obviously wrong.

(End of example)

A consistent requirement specification does not contain any conflicts and contradictions between individual requirement statements. The specified behavioral properties and constraints do not have an adverse impact on that behavior [132].

Inconsistent software models can have negative and positive effects in the software development life-cycle [131]:

- negative effects of inconsistency can influence negatively the timeliness, cost, safety and reliability, and maintenance.
- positive effects of inconsistency can invoke a deeper further analysis. However, it has to be appreciated that these benefits arise only if inconsistencies are allowed to emerge as models evolve. They can be tolerated for at least some period and used as drivers of managed interactions among the stakeholders that can deliver the above benefits [103].

The problem of inconsistencies in software models has been considered since the late eighties. There are techniques, methods and tools which support the identification, analysis, and treatment of various forms of inconsistencies in models expressed in a wide range of modeling languages and notations, including:

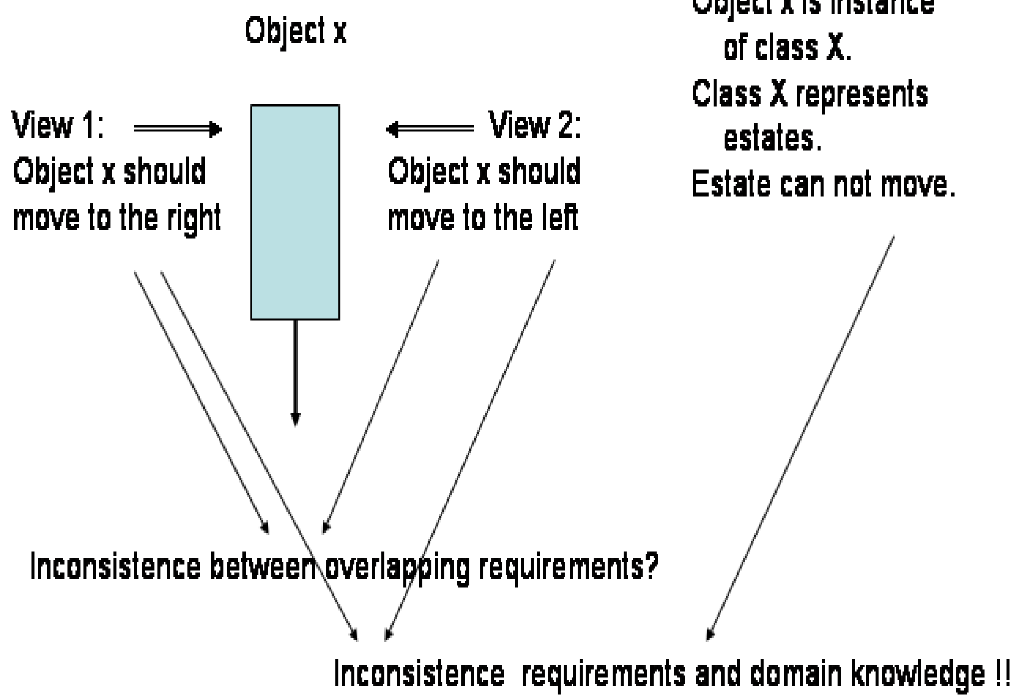

Figure 10.1: Inconsistence

- formal specification languages including first-order logic [49], [58], [107], [51], [129], language Z [21], [18], [148], KAOS [139], [140].
- structured requirements templates [120], [46], [91].
- state transition diagrams [63] and state-based languages [28].
- conceptual graphs [42].
- object-oriented languages UML [32], [127], [128], [148], [130].

We can view inconsistency management as a process composed of six activities as described in [131]. These activities are:

- detection of overlaps,
- detection of inconsistencies,
- diagnosis of inconsistencies,
- handling of inconsistencies,
- tracking of inconsistencies,
- specification and application of a management policy for inconsistencies.

The complete inconsistency management is out of the scope of our work. Because of the proposed method we address only detection of overlaps, detection of inconsistencies, and diagnosis of inconsistencies.

10.2.1 The Inconsistency Management Process

Inconsistency management has been defined by [60] as the process by which inconsistencies between software models are handled so as to support the goals of the stakeholders concerned. In literature, two general frameworks have been proposed describing the activities that constitute this process: one by Finkelstein [57] and one by Nuseibeh [111].

Both frameworks share the premise that an inconsistency in software models has to be established in relation to a specific consistency rule and that the process of managing inconsistencies includes activities for detecting, diagnosing and handling them.

These main activities are:

- **Detection of overlaps** This activity is performed to identify overlaps between the software models. The identification of overlaps is a crucial part of the overall process since models with no overlapping elements cannot be mutually inconsistent. The identification of overlaps is carried out by agent(s) which are specified in the inconsistency management policy.
- **Detection of inconsistencies** This activity is performed to check for violations of the consistency rules by the software models. The consistency rules are to be checked.
- **Diagnosis of inconsistencies** This activity is concerned with the identification of the source, the cause and the impact of an inconsistency.

- The source of an inconsistency is the set of elements of software models which have been used in the construction of the argument that shows that the models violate a consistency rule [110].
- The cause of an inconsistency is defined as the conflict(s) in the perspectives and/or the goals of the stakeholders which are expressed by the elements of the models that give rise to the inconsistency.
- The impact of an inconsistency is defined as the consequences that an inconsistency has for a system.

The source and the cause of an inconsistency have a very important role in the inconsistency management process since they can be used to determine what options are available for resolving or ameliorating an inconsistency and the cost and the benefits of the application of each of these options. Specifying the impact of an inconsistency in qualitative or quantitative terms is also necessary for deciding with what priority the inconsistency has to be handled and for evaluating the risks associated with the actions for handling the inconsistency which do not fully resolve it.

The handling and tracking of inconsistencies has been considered as a central activity in inconsistency management. But the inconsistency management process is out of the scope of this report.

10.2.2 Human Inspection

Jackson (1997) suggests the identification of overlaps by virtue of ”designations”. Designations constitute a way of associating non-ground terms in formal models with ground terms which are known to have reliable and unambiguous interpretations (called ”phenomena”). A designation associates a non-ground term with a recognition rule which identifies the phenomena designated by it. The rule is specified in natural language. Designations can certainly help stakeholders in identifying overlaps but should not be used as definitive indication of them. The reason is that the recognition rules of the designations might themselves admit different interpretations.

The idea to use functors to represent overlaps, has been supported by other authors [48] who, however, criticised the fact that it should be checked whether functors preserve the structures of the model parts they interconnect. This criticism has been on the grounds that such a check would be too strict in the development of large software systems.

Spanoudakis in [129] has also acknowledged the need to check the consistency of overlap relations but they propose a less strict check. According to them, a set of asserted overlap relations should be checked whether they satisfy certain properties which arise from the formal definition of overlaps. For example, if it has been asserted that a model element a inclusively overlaps with a model element b but has no overlap with a third element c then it should be checked that there is no total, inclusive or partial overlap between b and c. In their view, this check should be performed before checking the consistency of the models involved. This check is particularly useful in cases of overlaps asserted by humans.

The main difficulty with the identification of overlaps using inspections by humans is that this identification becomes extremely time consuming even for models of moderate complexity.

10.2.3 Similarity Analysis

The fourth general approach is to identify overlaps by automated comparisons between the models. This approach exploits the fact that modelling languages incorporate constructs which imply or strongly suggest the existence of overlap relations.

For instance, the ”Is-a” relation in various object-oriented modeling languages is a statement of either an inclusive overlap or a total overlap. This is because ”Is-a” relations normally have a set-inclusion semantics, that is the subtype designates a proper or not-proper subset of the instances of the supertype. Similarly, the implication ($\rightarrow$) between two predicates of the same arity constitutes a statement of inclusive overlap in a first order language. Overall, it should be noted that similarity analysis techniques tend to be sensitive to extensive heterogeneity in model representation, granularity and levels of abstraction.

10.3 Checking Completeness of Requirements Specification

10.3.1 Logical Completeness

One kind of completeness is logical completeness. It is a property associated with combining a procedure for constructing well-formed formulas (wffs), a definition of truth that relates to interpretations and models of logical systems, and a proof procedure that allows new wffs to be derived from old wffs. A logical system is logically complete if every true wff can be derived.

The other side to logical completeness is consistency. A logical system is inconsistent if it contains a contradiction. If falsity can be derived, then any wff can be derived, so trivially all true wffs can be derived.

10.3.2 Computational Completeness

A second kind of completeness is computational completeness, which is a property of a programming language. Consider the programming language called a Turing machine. No one has ever invented a programming language in which a program could be written that could not also be represented as a Turing machine. Indeed, the Church-Turing thesis is that it is impossible to have a programming language more powerful than a Turing machine. This thesis has not been proved, but it has never been seriously challenged.

Programming languages exist that are not complete (e.g. SQL). In other words, it is possible to formulate programs that these languages cannot represent. For example, SQL does not support recursion, which makes it unsuitable for a range of problems including bill-of-materials applications.

The other side to computational completeness is non-termination. Programs can be written that go into an infinite loop or fail to terminate for some more complex reason. It is impossible to write a terminating program, however, that tests other programs for termination. This outcome is called the unsolvability of the halting problem.

10.3.3 Completeness of Ontology

Both, logical and computational completeness, are standard results that are widely known [75]. Ontological completeness, however, is a topic of current specific research.

Both, logical completeness and computational completeness, are properties of the languages and reasoning platforms that are used to express statements, not properties of the statements themselves. We would therefore expect that ontological completeness was a property of the language and reasoning platform used to construct a software system. A view given by Debenham [41] bases maintainability (non-functional requirement) on a relationship between the semiotic system realised in the specification of the system and the semiotic system of the organisation

in which the software system is used. A semiotic system is a characterisation of a language in terms of its grammar, the collection of words in it, how the words are related to each other, and the conventions governing the use of the grammar in creating texts [40].

As we already mentioned in Section 1.2 the question of ontology completeness is out of the scope of this report. For our purpose, we at least want to say that a system is ontologically incomplete if it is not completely specified.

10.3.4 Completeness of Requirements

A complete requirements specification must precisely define all real world situations that will be encountered and the capabilitys responses to them (all use cases, all scenarios, etc.).

As written in Parnas [92], problem requirements should be a subset of domain konwledge. Having both of them described as ontologies we can decide using the reasoning system if it is true. We can find whether all concepts of the problem requirements are contained in the domain knowledge and the same we can decide on relations of problem requirements:

- If C is a set of concepts, a concept c2 has direct or transitive association with concept c1, and c1 $\in$ C, c2 $\notin$ C, then we call concepts set C incomplete. The solution is to add c2 to C.
- If concepts c1, c2 $\in$ C, and c1 has direct association with c1, i.e. r(c1, c2), but r $\notin$ R, so we call the relation set R incomplete. The solution is to add r to R.
- The analysis process of completeness is iterative, and will end when the result concepts and relations set are stable.

This means that completeness is detected according to inherent relation, i.e. by reasoning. If two requirements have inherent relation in domain knowledge base, the related requirement should be included in requirements set when the source requirement is selected. This process can improve completeness of requirements but cannot guarantee it, of course.

10.4 Checking Validity of Requirements Specification

Checking the validity of a requirements specification means to check how much the product based on the requirements satisfies the customer's expectations.

The problem is that some of the customer's expectations are not described in requirements specifications, i.e. the requirements are not complete, because the customer never mentioned them or because some other expectations are described in a different way in requirements specifications or because the analyst did not exactly understand the customer's needs and the customer did not understand exactly what the analyst wrote.

In standard software engineering we have the following possibilities:

- It can be decided about validity not until the customer starts using the product or at least the prototype. But this is too late because too much money has been invested into the product development in between and all changes are getting to expensive.
- We use the prototyping incremental development which means that the customer can use and validate some parts of the prototype, i.e.g. screens, very soon confronting his expectations with properties and features of the system under construction.

- Before having software components of the prototype implemented we use inspections and reviews of documents describing the first phase of the software development to discover problems.

Using the approach described in this report, the completeness of requirements can be improved because the requirements (converted into ontology) will be confronted with the domain knowledge (described as an ontology) and some missing parts can be identified as mentioned above. As a reaction the user will be asked whether he/she needs these parts of the domain knowledge being integrated in requirements.

10.5 Why to Use Ontology for Checking Requirements Specifications

Using ontologies to shape the requirements engineering process is clearly not a new idea. In the area of knowledge engineering, ontology was first defined by [102]. It defines ontology as the basic terms and relations comprising the vocabulary of a topic area, as well as the rules for combining terms and relations to define extensions to the vocabulary.

To address the problem of the lack of a shared understanding and the consequent poor communication among analyst and client, ontologies seem to be the right tool because they are designed to capture natural language descriptions of domains of interest, provide more consistent representations of such domains, reduce ambiguity inherent in communication, reduce error via the structuring of knowledge, and provide ways to extend and specialize captured domain knowledge.

The two sources of ontological categories are:

- Observation - An ontology is proposed as the support structure for a requirements development and modeling tool that could be used during user interviews, to improve communication between participants. An ontology necessarily contains some sort of world view with respect to a given domain. The world view is often conceived as a set of concepts (e.g. entities, attributes, processes), their definitions and their inter-relationships. This is referred to as a conceptualization. Here, ontology represents the domain knowledge and requirements can be seen as a specialized subset of it.

- Reasoning - Additionally, ontology not only defines a common vocabulary for persons who need to share information it also contains a logical theory that constrains the intended models of logical language containing:
  - integrity rules of the domain model representing the domain knowledge,
  - derivation rules and constraint rules of the problem model.

  Reasoning in ontologies brings the inferential capabilities that are not present in taxonomies.

Using ontologies supports consistency which is critical to the requirements engineering process. Consistent understanding of the domain of discourse reduces ambiguity and lessens the impact of contextual differences between participants.

Moreover, ontologies are uniquely positioned to be conductive to both users and programmers. There is a natural language part which is apprehensible for users and a logical, object-oriented part for programmers. They are re-usable and extendable to a variety of domains and therefore can be specialized for the needs of each new user [136].

10.5.1 Using Shared Ontologies in Knowledge Acquisition

Ontologies are used in a variety of ways. Those that are similar to the work of our proposal include [27]. They are using an ontology-based approach to knowledge acquisition from text through the use of natural language recognition. In [137] they have constructed the Enterprise Ontology to aid developers in taking an enterprise-wide view of an organisation which can then be used to aid in decision making, requirements specification, and communicating and sharing knowledge across an organization. The work [87] proposes a threetiered ontology to specifically guide the requirements elicitation process. This approach is intended to automate both interactions with users and the development of application models.

However, the work outlined here is not specifically addressing the issue of improving natural language communication between stakeholders in an interview in order to achieve more polished requirements. This report will investigate the possibility of combining UML model and OWL ontology for checking and validating requirements specifications, as we have already mentioned above.

A total overlap in this approach is assumed when two model elements are ”tagged” with the same item in the ontology [120].

The ontologies used by [120] in their Oz system are domain models which prescribe detailed hierarchies of domain objects, relationships between them, goals that may be held by the stakeholders, and operators which achieve these goals. The software models which can be handled by their techniques are constructed by instantiating the common domain models they propose.

In searching for inconsistencies, their techniques assume total overlaps between model elements which instantiate the same goal in the domain model. The ontology used by the QARCC system [16] is a decomposition taxonomy of software system quality attributes. This ontology also relates quality attributes with software architectures and development processes that can be used to achieve or inhibit them.

10.6 Weak points of related papers

As a weak point of related papers the fact can be stressed that there is no reasoning applied, at least not in such a range as in our proposal. The reason is that there were no technical means to do it at the time when the main part of this research was running.

Bibliography

[1] Agarwal, R., Tanniru, M.R.: Knowledge acquisition using structured interviewing: An empirical investigation. Journal of Management Information Systems, 7(1), pp. 123-140, 1990.

[2] Alcazar, E.G., Monzn, A.: A process framework for requirements analysis and specification. In: Proceedings of the 4th International Conference on Requirements Engineering, IEEE Computer Society, 2000.

[3] Alvarez, R.: Theres more to the story: Using narrative analysis to examine requirements engineering as a social process. Paper presented at the 7th International Workshop on Requirements Engineering, Interlaken, 2001.

[4] Alvarez, R.: Discourse analysis of requirements and knowledge elicitation interviews. In: Proceedings of the 35th Hawaii International Conference on System Sciences (HICSS02), IEEE Computer Society, 2002.

[5] Alvarez, R., Urla, J.: Tell me a good story: Using narrative analysis to examine information requirements interviews during an ERP implementation. The Data Base for Advances in Information Systems, 33(1), pp. 38-52, 2002.

[6] Anton, A.I., Earp, J.B.: A requirements taxonomy for reducing Web site privacy vulnerabilities. Requirements Engineering, 9(3), pp. 169-185, 2004.

[7] Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.(Eds.): The Description Logics Handbook: Theory, Implementation, and Application. Cambridge University Press, 2003.

[8] Bandler, R., Grinder, J: The Structure of Magic. Science and Behavior Books, 1990.

[9] Barbosa, S.D.J., et al.: Supporting a shared understanding of communication-oriented concerns in human-computer interaction: A lexicon-based approach. Lecture Notes in Computer Science, No. 3425, pp. 271-288, 2005.

[10] Berglas, A.: Non-monotonic Reasoning and End-User Conceptual Modelling. PhD thesis, The University of Queensland, 1997.

[11] Berry, D.M., Kamsties, E.: The Dangerous 'All' in Specifications. Proceedings of the Tenth International Workshop on Software Specification and Design (IWSSD'00), San Diego, IEEE, November 2000.

[12] Berry, D.M., Kamsties, E., Krieger, M.M.: From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity - A Handbook. In: Sampaio do Prado Leite, Julio Cesar; Doorn, Jorge Horacio (Eds.): Perspectives on Software Requirements, Series: The Springer International Series in Engineering and Computer Science , Vol. 753, Springer, 2003.

[13] Bettini, C.: A family of Temporal Terminological Logics. Advances in Artificial Intelligence: 3rd Congress of IAAI, Springer Verlag, LNCS No.728, 1993.

[14] Bharadwaj, R., Heitmeyer, C.: Model Checking Complete Requirements Specifications Using Abstraction. Automated Software Engineering, 6, pp. 37-68, 1995.

[15] Boehm, B.: Software Engineering Economics. Prentice Hall, 1981.

[16] Boehm B., In H.: Identifying Quality Requirements Conflicts. IEEE Software, pp. 25-35, March 1996, 1996.

[17] Boehm, B., Egyed, A.: Software Requirements Negotiation: Some Lessons Learned. In: Proceedings of the 20th International Conference on Software Engineering, 1998.

[18] Boiten, E., Derrick, J., Bowman, H., Steen, M.: Constructive Consistency Checking for Partial Specification in Z. Science of Computer Programming, 35(1), pp. 29-75, September 1999.

[19] Booch, G., Rumbaugh, J. and Jacobson, I.: The Unified Modeling Language user guide, Addison-Wesley, 1997.

[20] Borgida, A., PatelSchneider, P.F.: A semantics and complete algorithm for subsumption in the classic description logic. Journal of Artificial Intelligence Research, pp.277–308, 1994.

[21] Bowman, H., Derrick, J., Linington, P., Steen M.: Cross-viewpoint Consistency in Open Distributed Processing. IEE Software Engineering Journal, Vol. 11, No. 11, pp. 44-57, 1996.

[22] Brachman, R.J., Levesque, H.J.: The tractability of subsumption in framebased description languages. Proc. AAAI84, Austin, pp. 34–37, August, 1984.

[23] Brockmans, S., Volz, R., Eberhart, A.: Visual Modeling of OWL DL Ontologies Using UML [M]. LNCS Vol. 3298, pp.198-213, 2004.

[24] Bunge, M.: Treatise on Basic Philosophy: Volume 2: Semantics II: Interpretation and Truth. Reidel, Dordrecht, Holland 1974.

[25] Bunge, M.: Treatise on Basic Philosophy: Volume 3: Ontology I: The Furniture of the World. Reidel, Dordrecht, Holland 1977.

[26] Bunge, M.: Treatise on Basic Philosophy: Volume 4: Ontology II: A World of Systems. Reidel, Dordrecht, Holland 1979.

[27] Carreno, R.S., et al.: An ontology-based approach to knowledge acquisition from text. Cuadernos de Filologia Inglesa, 9(1), pp. 191-212 (in English), 2000.

[28] Chan, W. et al.: Model checking large software specifications. IEEE Transactions on Software Engineering, Vol 24, No. 7, pp. 498-520, 1998.

[29] CHAOS Report. The Standish Group, 1995.

[30] Chen, P.P.S.: The Entity-Relationship Model - Towards a Unified View of Data. ACM Transactions on Database Systems, 1(1), pp. 9-36, 1976.

[31] Christopherson, L.L.: Use of an ontology-based note-taking tool to improve communication between analysts and their clients. A Masters Paper for the M.S. in I.S.degree, University of North Carolina, November, 2005.

[32] Clarke, S., Murphy, J., Roantree, M.: Composition of UML Design Models: A Tool to Support the Resolution of Conflicts. In: Proceedings of the Int. Conference on Object-Oriented Information Systems, Springer, pp. 464-479, 1998.

[33] Colomb, R.M.: Impact of Semantic Heterogeneity on Federating Databases. The Computer Journal, 40(5), pp. 235-244, 1997.

[34] Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Softwar, pp. 37-45, November 1998.

[35] Coughlan, J., Macredie, R.D.: Effective communication in requirements elicitation: A comparison of methodologies. Requirements Engineering, 7(2), pp. 47-60, 2002.

[36] Cranefield, S., Purvis, M.: UML as an Ontology Modelling Language. In: Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 1999.

[37] Cranefield, S., Purvis, M.: A UML profile and mapping for the generation of ontology-specific content languages. The Knowledge Engineering Review, Volume 17, Issue 1, pp. 21-39, 2002.

[38] CS2 Software Engineering note, 2 CS2Ah, Autumn 2004. http://www.inf.ed.ac.uk/teaching/courses/cs2/LectureNotes/CS2Ah/SoftEng/se02

[39] Davis, A. M.: Software Requirements: Objects, Functions, and States. Englewood Cliffs, Prentice-Hall, 1993.

[40] de Saussure, F.: Course in General Linguistics. Duckworth, London, 1983.

[41] Debenham, J.K.: Knowledge Systems Design. Prentice Hall, 1989.

[42] Delugach, H.: Analyzing Multiple Views Of Software Requirements. In: Eklund, P., Nagle, T., Nagle, J., Gerholz, L. (eds.): Conceptual Structures: Current Research and Practice, Ellis Horwood, New York, pp. 391-410, 1992.

[43] Devanbu, P., Jones, M.: The use of description logics in KBSE systems. Proc. 17th Int. Conf. on Software Engineering, Sorrento, Italy, 1994.

[44] Djuric, D.: MDA-based Ontology Infrastructure. International Journal on Computer Science and Information Systems , Vol. 1, No. 1, pp. 91-116, 2004.

[45] http://projects.semwebcentral.org/projects/codip

[46] Easterbrook, S.: Handling Conflict between Domain Descriptions with Computer-Supported Negotiation. Knowledge Acquisition, 3, pp. 255-289, 1991.

[47] Easterbrook, S.: A Survey of Empirical Studies of Conflict, CSCW: Cooperation or Conflict. Easterbrook, S. (ed.), Springer-Verlag, pp. 1-68.

[48] Easterbrook, S., Callahan, J., and Wiels, V.: V & V Through Inconsistency Tracking and Analysis. Proceedings of International Workshop on Software Specification and Design, Kyoto, 1998.

[49] Easterbrook, S., Finkelstein, A., Kramer, J., Nuseibeh, B.: Co-Ordinating Distributed ViewPoints: the anatomy of a consistency check. International Journal on Concurrent Engineering: Research & Applications,2,3, CERA Institute, USA, pp. 209-222, 1994.

[50] Easterbrook, S., Nuseibeh, B.: Using ViewPoints for Inconsistency Management. IEE Software Engineering Journal, 1995.

[51] Easterbrook, S., Nuseibeh, B.: Managing Inconsistencies in an Evolving Specification. Proceedings of the 2nd International Symposium on Requirements Engineering, IEEE Press, pp. 48-55, 1995.

[52] Emmerich, W.: An Architecture for Viewpoint Environments based on OMG/CORBA. Joint Proceedings of the Sigsoft 96 Workshops Viewpoints 96, ACM Press, pp. 207-211, 1996.

[53] Emmerich, W., Finkelstein, F., Montangero, C., Antonelli, S., Armitage, S.: Managing Standards Compliance. IEEE Transactions on Software Engineering, 25, 6, 1999.

[54] Eriksson, H.: Using JessTab to integrate Protege and Jess. Intelligent Systems, Volume 18, Issue 2, pp. 43- 50, IEEE Mar-Apr 2003.

[55] http://www.eurostep.com/prodserv/exff/exff.html

[56] Falkovych, K.: Ontology Extraction from UML Diagrams. Master's thesis, Vrije Universiteit Amsterdam, 2002.

[57] Finkelstein, A., Spanoudakis, G., Till D.: Managing Interference. Joint Proceedings of the Sigsoft 96 Workshops Viewpoints 96, ACM Press, pp. 172-174, 1996.

[58] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling In Multi-Perspective Specifications. IEEE Transactions on Software Engineering, 20, 8, pp. 569-578, 1994.

[59] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: a framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering, 2, 1, pp. 31-58, 1992.

[60] Finkelstein, A., Sommerville I.: The Viewpoints FAQ. Software Engineering Journal, Vol. 11, No. 1, pp. 2-4, 1996.

[61] Galliers, R.D., Swan, J.A.: Theres more to information systems development than structured approaches: Information requirements analysis as a socially mediated process. Requirements Engineering, 5(2), pp. 74-82, 2000.

[62] http://www.sfu.ca/ dgasevic/projects/UMLtoOWL

[63] Glinz, M.: An Integrated Formal Model of Scenarios Based on Statecharts. Proceedings of the 5th European Software Engineering Conference (ESEC 95), LNCS 989, Springer-Verlag, pp.254-271, 1995.

[64] Goetz, R., Rupp, Ch.: Psychotherapy for System Requirements. Proceedings of the Second IEEE International Conference on Cognitive Informatics (ICCI03), 2003.

[65] Goguen, J.A., Linde, C.: Techniques for requirements elicitation. In: Proceedings of IEEE International Symposium on Requirements Engineering, IEEE Computer Society, pp. 152-164, 1993.

[66] Groves, L., et al.: A Survey of Software Requirements Specification Practices in the New Zealand Software Industry. In: Proceedings of the 2000 Australian Software Engineering Conference ASWEC'00, IEEE Computer Society, 2000.

[67] Gruber, A.: A translation approach to portable ontology specifications. Knowledge Acquisition, 5(2), pp.199-220, 1993.

[68] Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. International Journal of Human-Computer Studies, 43(5/6), pp. 907-928, 1995.

[69] Gruninger, M., Lee, J.: Introduction to the ontology application and design section. Guest editors-communications of the ACM, 45(2), pp. 39-41, Feb, 2002.

[70] Guarino, N.: Formal Ontology, Conceptual Analysis and Knowledge Representation. International Journal of Human and Computer Studies, 43(5/6), 1995.

[71] Guarino, N.: Understanding, building, and using ontologies. International Journal of Human-Computer Studies, 46(2/3), pp. 293-310, 1997.

[72] Haarslev, Moeller: Description of the RACER System and its Applications. Description Logics, pp. 132-141, Stanford, USA, 2001.

[73] Hands, K., Peiris, D., Gregor, P.: Development of a computer-based interviewing tool to enhance the requirements gathering process. Requirements Engineering, 9(3), pp. 204-216, 2004.

[74] Heflin, J. (ed.): OWL web ontology language use cases and requirements. 2004. http://www.w3.org/TR/webont-req/.

[75] Hofstadter, D.: Gödel, Escher, Bach: An Eternal Golden Braid. Penguin, 1980.

[76] Hong-wei, W., Jia-chun, W.,Fu, J.: A study on Ontology Model Based on Description logic. Systems Engineering, 21(3), pp.101-106, 2003.

[77] Horridge, M., et al.: A Practical Guide to Building OWL Ontologies Using the Protg-OWL Plugin and CO-ODE Tools, Edition 1.0. http://protege.stanford.edu/doc/users.html.

[78] Hunter, A., Nuseibeh, B.: Managing Inconsistent Specifications: reasoning, analysis and action. Department of Computing Technical Report Number 95/15, Imperial College, London, UK.

[79] Hunter, A., Nuseibeh, B.: Analysing Inconsistent Specifications. In: Proceedings of 3rd International Symposium on Requirements Engineering (RE 97), Annapolis, USA, January, IEEE CS Press, pp. 78-86, 1997.

[80] Hunter, A., Nuseibeh, B.: Managing Inconsistent Specifications: Reasoning, Analysis and Action. ACM Transactions on Software Engineering and Methodology, Vol. 7, No. 4, pp. 335-367, 1998.

[81] Jackson, M.: Problem Frames: Analyzing and Structuring Software Development Problems. Addison-Wesley, 2001.

[82] http://herzberg.ca.sandia.gov/jess/

[83] http://www.ida.liu.se/ her/JessTab/

[84] Jess 7.1 manual.
Sandia National Laboratories.
http://www.jessrules.com/jess/docs/index.shtml,2007.{OMG}
http://www.omg.org/techprocess/meetings/schedule/
UML_2.0_Superstructure_FTF.html

[85] Jiang, D., Zhang, S., Wang, Y.: Towards a formalized ontology-based requirements model. Journal of Shanghai Jiaotong University (Science), 10(1), pp. 34-39, 2005.

[86] Jin, Zhi: Ontology-based requirements elicitation automatically. Chinese J. Computers, 23(5), pp. 486-492, May 2000.

[87] Jin, Z.: Ontology-Based Requirements Elicitation. Journal of Computers, 23(5), pp. 486-492, 2003.

[88] Jin, Z., Bell, D.A., et al.: Automated requirements elicitation: Combining a model-driven approach with concept reuse. International Journal of Software Engineering and Knowledge Engineering, 13(1), pp. 53-82, 2003.

[89] Kaiya, H., Saeki, M.: Using Domain Ontology as Domain Knowledge for Requirements Elicitation. In: Proceedings of 14th IEEE International Requirements Engineering Conference, Minnesota, pp. 186-195, 2006.

[90] Kamsties, E., Berry, D.M., Paech, B.: Detecting Ambiguities in Requirements Documents Using Inspections. Workshop on Inspections in Software Engineering (WISE'01), pp. 68-80, Software Quality Research Lab, McMaster University, 2001.

[91] Kotonya, G., Sommerville, I.: Requierements Engineering with Viewpoints. Software Engineering Journal, 11(1), pp. 5-18, 1999.

[92] Kreyman, K., Parnas, D.L.: On Documenting the Requirements for Computer Programs Based on Models of Physical Phenomena. models3august, pp. 1-14, 2002.

[93] Kroha,P., Strauss,M.: Requirements Specification Iteratively Combined with Reverse Engineering. In: Plasil,F., Jeffery,K. (Eds.), SOFSEM'97: Theory and Practice of Informatics. Milovy, November 1997, Lecture Notes in Computer Science, No. 1338, Springer, 1997.

[94] Kroha,P.: Preprocessing of Requirements Specification. In: Ibrahim, M., King, J., Revell, N. (Eds.): Proceedings of the 11th International Conference Database and Expert Systems Applications DEXA 2000, London, Lecture Notes in Computer Science, No. 1873, Springer, 2000.

[95] Lee, Y., Zhao, W: An Ontology-based Approach for Domain Requirements Elicitation and Analysis. In: Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences IEEE. Hangzou, China, pp. 364- 371, 2006.

[96] Li, Z., Wang, Z., Zhang, A., Xu, Y.: The Domain Ontology and Domain Rules Based Requirements Model Checking. International Journal of Software Engineering and Its Applications, Vol. 1, No. 1, July, 2007.

[97] Marakas, G.M., Elam, J.J.: Semantic structuring in analyst acquisition and representation of facts in requirements analysis. Information Systems Research, 9(1), pp. 37-63, 1998.

[98] Matheus, Ch.J., Baclawski, K., Kokar, M.M.: BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules. Proceedings of the Second International Conference on Rules and Rule Markup Languages for the Semantic Web (RuleML'06), 2006.

[99] Mizoguchi, R., Ikeda, M.: Towards ontology engineering. In: Proceedings of the Joint 1997 Pacific Asian Conference on Expert Systems / Singapore International Conference on Intelligent Systems, pp. 259-266, 1997.

[100] Mikyeong Moon, Kenuhyuk Yeom, Heung Seok Chae: An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. IEEE Transactions on Software Engineering, 31(7), pp. 551-569, 2005.

[101] Nebel, B.: Terminological reasoning is inherently intractable. Artificial Intelligence 43, pp.235249, 1990.

[102] Neches, R., Fikes, R.E., Finin, T.: Enabling technology for knowledge sharing. AI Magazine, 12 (3), pp. 36-56, 1991.

[103] Nissen H., Jeusfeld M., Jarke M., Zemanek G., Huber H.: Managing Multiple Requirements Perspectives with Metamodels. IEEE Software, March 1996, pp. 37-47.

[104] Noy, N., McGuinness, D.L.: Ontology development 101: A guide to creating your first ontology. 2004. http://protege.stanford.edu/publications/ ontology-development/ontology101-noymcguinness.html.

[105] Nuseibeh, B.: Towards a Framework for Managing Inconsistency Between Multiple Views. Proceedings Viewpoints 96: International Workshop on Multi-Perspective Software Development, ACM Press, pp. 184-186, 1996.

[106] Nuseibeh, B., Finkelstein, A.: Viewpoints: a vehicle for method and tool integration. Proc. 5th International Workshop on CASE - CASE 92, IEEE CS Press, pp. 50-60, 1992.

[107] Nuseibeh, B., Finkelstein, A., Kramer, J.: Method Engineering for Multi-Perspective Software Development. Information and Software Technology Journal, 1994.

[108] Nuseibeh, B., Kramer, J., Finkelstein, A.: A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. IEEE Transactions on Software Engineering, Vol. 20, No. 10, pp. 760-773, 1994.

[109] Nuseibeh, B., Russo, A.: Using Abduction to Evolve Inconsistent Requirements Specifications. Austrialian Journal of Information Systems, 7(1), Special Issue on Requirements Engineering, 1999.

[110] Nuseibeh, B., Easterbrook, S., Russo, A.: Leveraging Inconsistency in Software Development. IEEE Computer, April 2000.

[111] Nuseibeh, B., Easterbrook, S.: Requirements Enginnering: a road map. In: Proceedings of ICSE'2000 - Future of Software Engineering Track, pp. 35-46, 2000.

[112] http://www.itee.uq.edu.au/ colomb/Papers/UML-OWLont04.03.01.pdf

[113] OWL Web Ontology Language Overview.
W3C Proposed Recommendation 15, December 2003, http://www.w3.org/TR/2003/PR-owl-features-20031215/

[114] Parnas, D.L., Asmis, G.J.K., and Madey, J.: Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety 32(2), pp. 189198, AprilJune 1991.

[115] Pomerantz, J.: A Linguistic Analysis of Question Taxonomies. Journal of the American Society for Information Science and Technology, 56(7), pp. 715-728, 2005.

[116] http://protege.stanford.edu/overview/index.html

[117] http://www.racer-systems.com/products/racerpro

[118] Robinson, M., Bannon, L.: Questioning representations. In: Proceedings of ECSCW, Amsterdam. http://www.ul.ie/ idc/library/papersreports/ LiamBannon/15/QuestFin.html.

[119] Robinson, W.: Interactive Decision Support for Requirements Negotiation. Concurrent Engineering: Research & Applications, 2, pp. 237-252, 1994.

[120] Robinson, W., Fickas, S.: Supporting Multiple Perspective Requirements Engineering. In: Proceedings of the 1st International Conference on Requirements Engineering (ICRE 94), IEEE Computer Society Press, pp.206-215, 1994.

[121] Robinson, W.: I Didnt Know My Requirements were Consistent until I Talked to My Analyst. Proceedings of 19th International Conference on Software Engineering (ICSE-97), IEEE Computer Society Press, Boston, USA, May , pp. 17-24, 1997.

[122] Robinson, W., Pawlowski, S.: Managing Requirements Inconsistency with Development Goal Monitors. IEEE Transactions on Software Engineer, November/December, 1999.

[123] Rosenberg, D., Scott, K.: Use Case Driven Object Modeling With UML: A Practical Approach. Addison-Wesley, Reading, Mass., 1999.

[124] Ru Qian Lu, Jin Zhi, Chen Gang: Ontology-based requirements analysis. Journal of Software, 11(8), pp. 1009-1017, 2000.

[125] Sadiq, W., Orlowska, M.: On Correctness Issues in Conceptual Modeling of Workflows. Proceedings of the 5th European Conference on Information Systems, Cork, Ireland, pp. 943-964, 1997.

[126] Spanoudakis, G., Constantopoulos, P.: Integrating Specifications: A Similarity Reasoning Approach. Automated Software Engineering Journal, Vol. 2, No. 4, pp. 311-342, 1995.

[127] Spanoudakis, G., Finkelstein, A.: Reconciling Requirements: a method for managing interference, inconsistency and conflict. Annals of Software Engineering, Special Issue on Software Requirements Engineering, 1997.

[128] Spanoudakis, G., Finkelstein, A.: A Semi-automatic process of Identifying Overlaps and Inconsistencies between Requirement Specifications. In: Proceedings of the 5th International Conference on Object-Oriented Information Systems (OOIS 98), pp. 405-424, 1998.

[129] Spanoudakis, G., Finkelstein, A., Till, D.: Overlaps in Requirements Engineering. Automated Software Engineering Journal, Vol 6, pp. 171-198, 1999.

[130] Spanoudakis, G., Kassis, K.: An Evidential Framework for Diagnosing the Significance of Inconsistencies in UML Models. Proceedings of the International Conference on Software: Theory and Practice, World Computer Congress 2000, Bejing, China, pp. 152-163, 2000.

[131] Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. In: Chang, S.K. (ed.): Handbook of Software Engineering and Knowledge Engineering, World Scientific Publishing Co., pp. 329-380, 2001.

[132] Stokes, D.A.: Requirements analysis. Computer Weekly Software Engineers Reference Book, 1991.

[133] Sutcliffe, A., Maiden, N.: The domain theory for requirements engineering. IEEE Transactions on Software Engineering, 24(3), pp. 174-196, 1998.

[134] Sutton, D.: Linguistic problems with requirements and knowledge elicitation. Requirements Engineering, 5(2), pp. 114-124, 2000.

[135] http://protege.stanford.edu/doc/pdk/plugins/tab_widget.html

[136] Uschold, M., Gruninger, M.: Ontologies: Principles, methods, and applications. Technical Report AIAI-TR-191, Artificial Intelligence Application Institute, University of Edinburgh. Retrieved August 21, 2005 from http://www.aiai.ed.ac.uk/project/enterprise/ontology.html.

[137] Uschold, M., King, M., Moralee, S., Zorgios, Y.: The Enterprise Ontology. Knowledge Engineering Review, 13(1), pp. 31-89, 1998.

[138] van Lamsweerde, A.: Divergent Views in Goal-Driven Requirements Engineering. In: Joint Proceedings of the Sigsoft 96 Workshops Viewpoints 96, ACM Press, pp. 252-256, 1996.

[139] van Lamsweerde, A., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, November.

[140] van Lamsweerde, A., Letier, E.: Handling Obstacles in Goal-Oriented Requirements Engineering. IEEE Transcactions on Software Engineering, Vol 6, Special Issue on Exception Handling, 2000.

[141] Walling, C.: A Requirements Sublanguage for Automated Analysis. International Journal of Intelligent Systems, 10 (7), pp. 665-689, 1995.

[142] Weber, R.: Ontological Foundations of Information Systems. Coopers & Lybrand, 1997.

[143] Westrup, C.: Knowledge, legitimacy and progress? Requirements as inscriptions in information systems development. Information Systems Journal, 9(1), pp. 35-54, 1999.

[144] Wilson, W.M., et. al.: Automated Analysis of Requirement Specifications. Proceedings of the International Conference on Software Engineering (IASTED), 1997.

[145] W. A. Woods, Schmolze, J.G.: The KLONE family. Computers and Mathematics with Applications 23(25), Special Issue on Semantic Networks in Artificial Intelligence, 1991.

[146] Zhangwei, Mei Hong: A feature-oriented domain model and its modeling process. Journal of Software, 14(8), pp. 1345-1356, 2003.

[147] Zhu, X., Zhi, Jin: Inconsistency Measurement of Software Requirements Specifications an Ontology-Based Approach. In: Proceedings of the 10th IEEE International Conference on engineering of Complex Computer Systems, 2005.

[148] Zisman, A., Emmerich, W., Finkelstein, A.: Using XML to Specify Consistency Rules for Distributed Documents. In: 10th International Workshop on Software Specification and Design (IWWSD-10), Shelter Island, San Diego, California, November 2000.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-04-01 Karsten Hilbert, Guido Brunnett, A Hybrid LOD Based Rendering Approach for Dynamic Scenes, Januar 2004, Chemnitz

CSR-04-02 Petr Kroha, Ricardo Baeza-Yates, Classification of Stock Exchange News, November 2004, Chemnitz

CSR-04-03 Torsten Hoefler, Torsten Mehlan, Frank Mietke, Wolfgang Rehm, A Survey of Barrier Algorithms for Coarse Grained Supercomputers, Dezember 2004, Chemnitz

CSR-04-04 Torsten Hoefler, Wolfgang Rehm, A Meta Analysis of Gigabit Ethernet over Copper Solutions for Cluster-Networking, Dezember 2004, Chemnitz

CSR-04-05 Christian Siebert, Wolfgang Rehm, One-sided Mutual Exclusion A new Approach to Mutual Exclusion Primitives, Dezember 2004, Chemnitz

CSR-05-01 Daniel Beer, Steffen Höhne, Gudula Rünger, Michael Voigt, Software- und Kriterienkatalog zu RAfEG - Referenzarchitektur für E-Government, Januar 2005, Chemnitz

CSR-05-02 David Brunner, Guido Brunnett, An Extended Concept of Voxel Neighborhoods for Correct Thinning in Mesh Segmentation, März 2005, Chemnitz

CSR-05-03 Wolfgang Rehm (Ed.), Kommunikation in Clusterrechnern und Clusterverbundsystemen, Tagungsband zum 1. Workshop, Dezember 2005, Chemnitz

CSR-05-04 Andreas Goerdt, Higher type recursive program schemes and the nested pushdown automaton, Dezember 2005, Chemnitz

CSR-05-05 Amin Coja-Oghlan, Andreas Goerdt, André Lanka, Spectral Partitioning of Random Graphs with Given Expected Degrees, Dezember 2005, Chemnitz

CSR-06-01 Wassil Dimitrow, Mathias Sporer, Wolfram Hardt, UML basierte Zeitmodellierung für eingebettete Echtzeitsysteme, Februar 2006, Chemnitz

CSR-06-02 Mario Lorenz, Guido Brunnett, Optimized Visualization for Tiled Displays, März 2006, Chemnitz

CSR-06-03 D. Beer, S. Höhne, R. Kunis, G. Rünger, M. Voigt, RAfEG - Eine Open Source basierte Architektur für die Abarbeitung von Verwaltungsprozessen im E-Government, April 2006, Chemnitz

CSR-06-04 Michael Kämpf, Probleme der Tourenbildung, Mai 2006, Chemnitz

CSR-06-06 Torsten Hoefler, Mirko Reinhardt, Torsten Mehlan, Frank Mietke, Wolfgang Rehm, Low Overhead Ethernet Communication for Open MPI on Linux Clusters, Juli 2006, Chemnitz

Chemnitzer Informatik-Berichte

CSR-06-07 Karsten Hilbert, Guido Brunnett, A Texture-Based Appearance Preserving Level of Detail Algorithm for Real-time Rendering of High Quality Images, August 2006, Chemnitz

CSR-06-08 David Brunner, Guido Brunnett, Robin Strand, A High-Perpormance Parallel Thinning Approach Using a Non-Cubic Grid Structure, September 2006, Chemnitz

CSR-06-09 El-Ashry, Peter Köchel, Sebastian Schüler, On Models and Solutions for the Allocation of Transportation Resources in Hub-and-Spoke Systems, September 2006, Chemnitz

CSR-06-10 Raphael Kunis, Gudula Rünger, Michael Schwind, Dokumentenmanagement für Verwaltungsvorgänge im E-Government, Oktober 2006, Chemnitz

CSR-06-11 Daniel Beer, Jörg Dümmler, Gudula Rünger, Transformation ereignisgesteuerter Prozeßketten in Workflowbeschreibungen im XPDL-Format, Oktober 2006, Chemnitz

CSR-07-01 David Brunner, Guido Brunnett, High Quality Force Field Approximation in Linear Time and its Application to Skeletonization, April 2007, Chemnitz

CSR-07-02 Torsten Hoefler, Torsten Mehlan, Wolfgang Rehm (Eds.), Kommunikation in Clusterrechnern und Clusterverbundsystemen, Tagungsband zum 2. Workshop, Februar 2007, Chemnitz

CSR-07-03 Matthias Vodel, Mirko Caspar, Wolfram Hardt, Energy-Balanced Cooperative Routing Approach for Radio Standard Spanning Mobile Ad Hoc Networks, Oktober 2007, Chemnitz

CSR-07-04 Matthias Vodel, Mirko Caspar, Wolfram Hardt, A Concept for Radio Standard Spanning Communication in Mobile Ad Hoc Networks, Oktober 2007, Chemnitz

CSR-07-05 Raphael Kunis, Gudula Rünger, RAfEG: Referenz-Systemarchitektur und prototypische Umsetzung - Ausschnitt aus dem Abschlussbericht zum Projekt "Referenzarchitektur für E-Government" (RAfEG) -, Dezember 2007, Chemnitz

CSR-08-01 Johannes Steinmüller, Holger Langner, Marc Ritter, Jens Zeidler (Hrsg.), 15 Jahre Künstliche Intelligenz an der TU Chemnitz, April 2008, Chemnitz

CSR-08-02 Petr Kroha, José Emilio Labra Gayo, Using Semantic Web Technology in Requirements Specifications, November 2008, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz