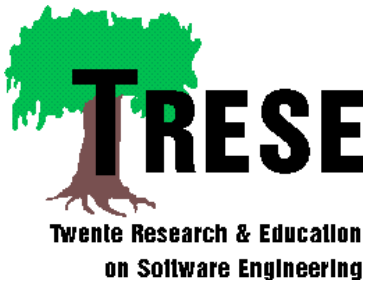

University of Twente
Enschede - The Netherlands

~ Master's thesis ~

Tool support for a metamodeling approach for reasoning about requirements

Jan-Willem Veldhuis

April 21, 2009

Comittee
A. Göknil, MSc.
dr. I. Kurtev, MSc.
dr.ir. K.G. van den Berg
prof.dr.ir. M. Akşit

Research group
University of Twente
Faculty of Electrical Engineering,
Mathematics and Computer Science
Software Engineering

Abstract

In the Software Engineering practice, software requirements are one of the earliest artifacts describing a system. Without requirements we cannot verify the quality of a delivered software product. Requirements are mostly textual descriptions. Traceability is considered essential to manage consistency between software development artifacts. Many research focused on the relation between requirements and other artifacts such as design, code and test cases. However, less attention has been paid to the relation *between* requirements.

Göknil et al. proposed a requirements metamodel. This provides structure to requirements models. This metamodel is distilled from key entities from several requirements engineering approaches described in literature. The main focus of the requirements metamodel is on requirements relations and their types. Furthermore, they provided formal semantics of the requirements relations in first-order logic. This enables reasoning on requirements and consistency checking. To provide a proof of concept for the metamodeling approach proposed by Göknil et al. we need an environment to model requirements conforming to the requirements metamodel. And we need a tool that supports first-order logic reasoning over requirements relations.

To the best of our knowledge, no requirements management tools exists which are capable of reasoning about requirements relations using formal semantics. Therefore, we developed a tool named TRIC (Tool for Requirements Inference and Consistency checking). TRIC is developed as an Eclipse RCP application. Requirements models are expressed in the Web Ontology Language (OWL), because first-order logic reasoners for OWL exist. Requirements models are stored and retrieved in RDFS/XML notation, enabling interoperability. To establish inference of implicit relations and to enable consistency checking we created a mapping between the formalization of requirements relations to OWL syntax and reasoner rules.

We evaluated TRIC using an example case of a Course Management System (CMS). We used the requirements for the tool to verify the design and implementation. The modeling of requirements in models conforming to the requirements metamodel is supported. The inference of implicit relations and consistency checking of the model is supported. The analysis of implicit relations is supported by a visualization engine. We investigated the scalability of the tool by looking at the time and resource behavior. The time needed for consistency checking increases exponentionally with the number of model elements. The memory consumption seems linear with respect to the model size.

TRIC does not support multiple metamodels. More research is needed on customizing the requirements metamodel and the formalization of additional relations. The inconsistencies found by the tool are not related to the contradicting requirements relations. TRIC supports the modeling and analysis of a requirements model. The next step is to support change impact analysis.

Acknowledgements

I would like to thank the people who helped me to realize this work.

Arda Göknil, Ivan Kurtev and Klaas van den Berg for guiding me through the project by giving constructive feedback. During the course *Advanced Design of Software Architectures* I got enthousiastic about a research project in the field of Model-Driven Engineering.

I would like to thank the hard-working guys of room 5070 (now relabeled to 5066) for a great atmosphere. It motivated me to carry on with the project. In particular I appreciate the help of David ten Hove with creating the example case of the Course Management System and Tjerk Wolterink for exchanging development experience with the Eclipse Rich Client Platform.

Especially I would thank Alfons Laarman for reviewing my thesis. I wish him a lot of fun and good luck with his PhD research.

Finally, I thank Karin for supporting me and of course my parents for their continuous support.

Jan-Willem Veldhuis, April 2009, Enschede

Contents

List of Figures

List of Abbreviations

AD	Architectural Design
CMS	Course Management System
EMF	Eclipse Modeling Framework
FOL	First-order logic
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MOF	Meta Object Facility
MVC	Model-View-Controller
OMG	Object Management Group
OWL	Web Ontology Language
OWL-DL	Description Logics variant of OWL
RCP	Rich Client Platform (of the Eclipse Framework)
RDF	Resource Description Framework
RE	Requirements Engineering
REMM	Requirements Engineering MetaModel
SWEBOK	Software Engineering Body of Knowledge
TRIC	Tool for Requirement Inference and Consistency checking
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

1.1 Introduction

In the process of Software Engineering, gathering requirements is one of the first steps. In order to develop a successful software product these requirements should be precise, complete and consistent. And because requirements happen to change over time, proper management of requirements is essential. Overlooking inconsistencies in an early phase could cause major problems when detected in a final stage. This is because requirements are related to the architectural design of a system. When requirements change the design and implementation of the system must change accordingly to meet the new demands.

The QuadREAD research project aims at bridging the gap between Requirements Engineering (analysts) and Architectural Design (architects). The analysis phase is one of the early phases in software development. Improvement of its quality, will improve overall quality, and will save time and money in later phases of development, testing and maintenance. Requirements evolve over time. One of the aims of the QuadREAD project is to elaborate on *traceability* research and focus on the tracing between requirements and architectural design decisions.

In Figure 1.1 an overview of relations between artifacts are shown. Requirements are established based on analysis of business demands. Requirements are realized in the Architectural Design (AD) . The AD is realized in detailed designs which are implemented in Systems.

Our work concentrates on the modeling of requirements (marked with the dashed ellipse in the figure).

1.2 Problem statement

Formalizations (of both requirements and their relations) are necessary for an automated and systematical approach to check requirements for consistency,

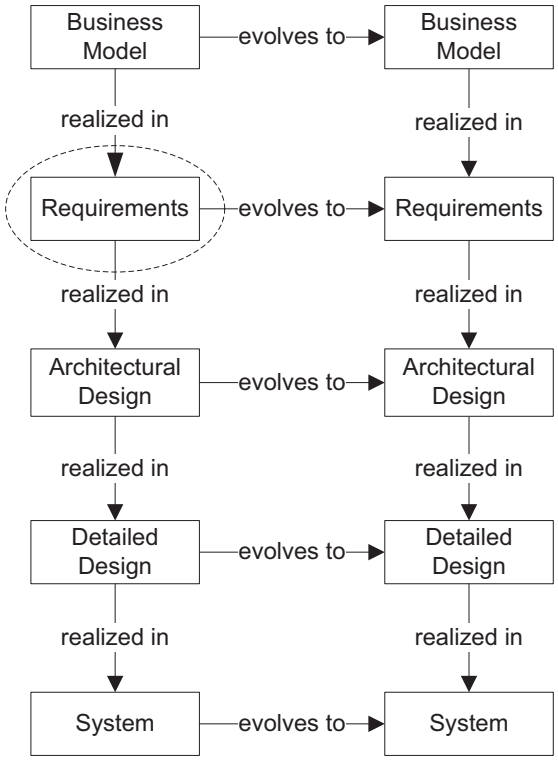

Figure 1.1: Traceability in system development

and to be able to analyze the impact of changing requirements. In practice several approaches are used to deal with requirements, however, they share a common problem: the requirements usually are textual descriptions. Relations between requirements are often implicit and hard to extract.

Göknil et al. [19] propose a metamodeling approach to capture requirements in models conforming to a *requirements metamodel*. By making relations between requirements explicit, automated reasoning can be applied to infer implicit relations and to detect inconsistencies. The approach however is not yet validated in an industrial context. Tool support is essential to conduct case studies and provide insight in the pragmatics of the approach.

However, to the best of our knowledge no tools exist which are able to model requirements conforming to a predefined metamodel, and are able to reason on different relation types to infer relations and detect inconsistencies. Therefore, the central research question is:

> **Can a tool be developed to support modeling of and automated reasoning on requirements models in order to infer implicit requirements relations and detect inconsistencies?**

This question can be divided in a number of sub questions:

- What are the requirements for the tool?
- How can requirements models be represented to conform to a metamodel and to enable reasoning?

- Which reasoner engine can be used to reason on requirements models?
- How do we create a mapping between the formalization of requirements and requirements relations to reasoner rules?
- How well does the resulting tool support the requirements reasoning approach?

1.3 Approach

In this work we will use the *core requirements metamodel* of Göknil et al. [19] to structure requirements and their relations. We will investigate different approaches to reason on requirements relations and design and implement a tool prototype.

First we explore the context of our work: requirements metamodels and relations between requirements. Capabilities of existing requirements tools with respect to requirements relations and reasoning are reviewed.

We study the formalization of requirement and requirements relations proposed by Göknil et al. in order to investigate the requirements for the tool support.

A number of requirements for the tool is defined. We aim at a proof of concept for the inference of implicit requirements relations and checking the modeled requirements for consistency.

We design and implement a tool prototype capable of handling both requirements models conforming to the requirements metamodel, and reasoning about requirements relations. To achieve this we create a mapping of the formalization of requirements to a set of reasoner rules.

To validate the tool support we constructed an example case: a basic requirements specification document for a Course Management System. We use all the relationship types defined in the metamodel and we intentionally added contradicting requirements in order to verify the consistency checking mechanism of the tool.

Using the example case and the set of requirements for the tool we evaluate the design and implementation of the tool.

1.4 Contributions

This thesis provides the following contributions:

> *"Tool support for the modeling of requirements conforming to the core requirements metamodel."*

In Chapter 5 we describe the requirements of the tool. In Chapter 6 we describe the design and Chapter 7 describes the implementation of the tool. In Chapter 8 we evaluate the tool.

> *"A mapping of the requirements formalizations by Goknil et al. [19] to a set of reasoner rules."*

The formalization of requirements and requirements relations is described in Chapter 4. In the chapter describing the implementation, Chapter 7, we describe how we mapped the formalization to a set of reasoner rules.

"Tool supported analysis of derived requirements relations."

In Chapter 7 we describe how we use the capability of the reasoner to analyze the derivation of an inferred requirements relation. This supports the requirements engineer to understand why and how the tool inferred a certain implicit relation.

1.5 Outline of this Thesis

In Figure 1.2 a map of this thesis is provided. It illustrates the relations between the chapters.

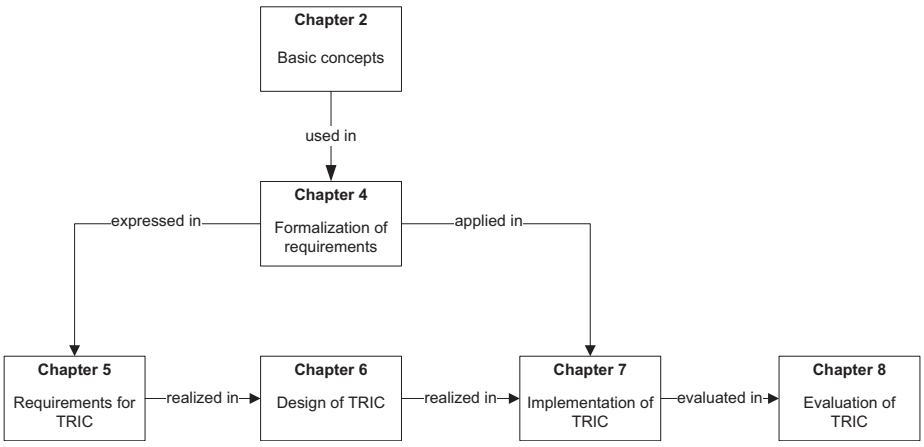

Figure 1.2: Map of this thesis

This thesis consists out of the following chapters:

Chapter 2 introduces some basic concepts used in this thesis, which are essential for understanding the work.

Chapter 3 elaborates on ongoing research on requirement metamodeling approaches and reasoning on requirements. Also provides an overview of reasoning capabilities of existing requirements tools.

Chapter 4 describes the formalization of requirements and requirement relations.

Chapter 5 describes the requirements and scope of the needed tool support.

Chapter 6 describes the design of the tool.

Chapter 7 describes the implementation of the developed tool.

Chapter 8 evaluates the tool, using the example case study and the requirements defined in Chapter 5.

Chapter 9 concludes and reflects on this work and discusses directions for future work.

2
Basic concepts

2.1 Introduction

This chapter introduces the basic concepts used in this thesis. In literature concepts can have multiple definitions. We describe the definitions we used in this work.

In Section 2.2 we introduce Model-Driven Engineering. Section 2.3 describes Requirements Engineering and the definition of a requirement we use in this work.

2.2 Model-Driven Engineering (MDE)

Model-Driven Architecture (MDA) is a software development approach. It considers models as first class entities for development (instead of program code). It was launched in 2001 by the Object Management Group (OMG) . It provides a guide for the structuring of specifications (models) [30].

One of the main aims of the MDA approach is to improve portability. It classifies models into two classes: Platform Independent Models (PIMs) and Platform Specific Models (PSMs). By means of a model transformation, a PIM can be transformed into a PSM.

Model-Driven Engineering (MDE) is an enhancement of MDA. It combines process and analysis with architecture [23]. Models can describe various concerns such as functionality, maintainability, security, etc. It can be compared with the usage of different blueprints (viewpoints) in civil engineering for carpenters, plumbers and electricians. Models abstract from reality and help understanding, communicating and analysis.

In [24] a model is defined as *"a description of (a part of) a system in a well-defined language"*.

Within the context of MDE a metamodel is constructing a formal definition of a model. Kurtev defines it as "a model of a modeling language" [26]. The

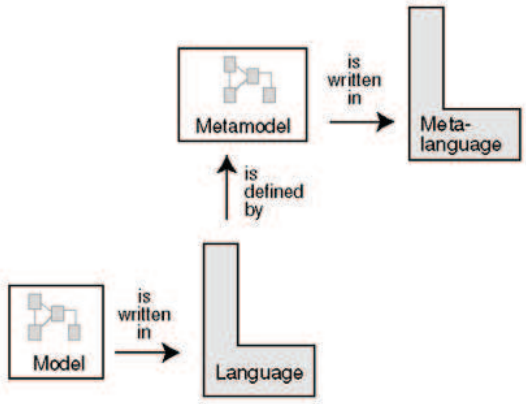

Figure 2.1: Models, languages, metamodels, and metalanguages [24]

relation between models, languages and metamodels is depicted in Figure 2.1. A metamodel constraints the structure of a model. When a model respects this structure, the model is said to *conform* to the metamodel.

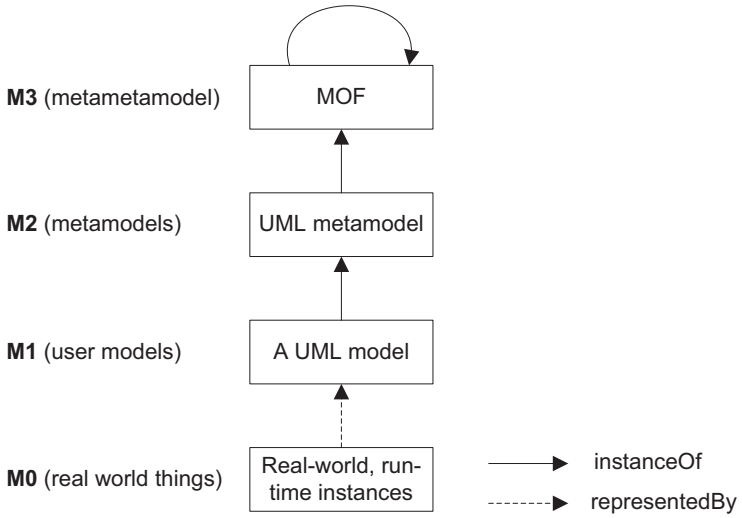

Figure 2.2: The four-layered architecture proposed by OMG

A metamodel is also a model and can be described in a modeling language. It conforms to its own metamodel, the metametamodel. This results in a hierarchy of models that spans multiple levels. The MDA standard defines a four-layer architecture, as depicted in Figure 2.2. The topmost layer (M3) is called the metametamodel. The OMG standard for the metametamodel is the Meta Object Facility (MOF). This layer is modeled in itself.

2.3 Requirements Engineering

In [34] an overview of requirements engineering (RE) is presented. They chose the following definition of Requirements Engineering defined by Zave [41]:

> "Requirements engineering is the branch of software engineering concerned with the realworld goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families."

The *realworld goals* represent the 'why' and 'what' of a system. The *precise specifications* are essential for analysis, validation and verification.

According to the IEEE's Software Engineering Body of Knowledge (SWEBOK) , a software requirement is *"a property which must be exhibited by software developed or adapted to solve a particular problem"* [2].

Requirements are often divided into two groups: functional and non-functional requirements:

functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities [2].

non-functional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements [2].

In this report we focus on the relations between requirements. As noted in the previous section a requirement is a property which must be exhibited by a system. We define a requirement formally as a tuple $\langle P, S \rangle$ where P is the property and S is the set of systems that satisfy P (i.e. $\forall s \in S : P(s)$). In Chapter 4 we provide more details about the formalization of requirements and requirements relations.

2.4 Conclusion

In this chapter we introduced the basic concepts used in this work. In Section 2.2 we described Model-Driven Engineering (MDE) as a software development approach. It raises the level of abstraction by considering models as first class entities as opposed to program code. Models are instances of metamodels. The Model-Driven Architecture, as launched by the Object Management Group (OMG), defines a four-layer architecture. The top-level is self-reflective, it is modeled in itself. Metamodels provide structure. This structure is useful to automate transformations, and for generic tool support.

In Section 2.3 we described requirements engineering. In our work we consider a requirement as a single artifact. We will focus on the relations between requirements without considering the contents of a requirement.

In the next chapter we discuss related research done on requirement relations, requirements metamodels and reasoning about requirements.

3
Related Work

3.1 Introduction

In this chapter we describe related work. We describe the state of the art of four topics: requirements relations, requirements metamodeling, reasoning about requirements and tool support.

In Section 3.2 we describe relevant research done on requirements relations. In Section 3.3 we describe attempts to define requirements metamodels. Section 3.4 describes the state of the research on reasoning about requirements. In Section 3.5 we describe the current tool support for requirements relations.

3.2 Requirements relations

Dahlsted et al. [10] stress the need to take requirements relations into consideration in order to make sound decisions during the software development process. However there is little known about these requirements relations. Their aim is to identify which types of requirements relations are critical to take into consideration in specific development situations.

The approach in the paper is to provide an overview of the requirements relation types (which they call 'requirements interdependencies') identified in current research.

Dahlstedt et al. conclude that there are several different types of requirements relations presented in literature focussing on different activities or development situations. They compiled the different views on requirements into an integrated model (Figure 3.1). The fundamental relation types are divided in two categories: Structural and Cost/Value.

With *structural* is meant that the relationships are of a hierarchical nature as well as of cross-structure nature. *Cost/value* relationships are concerned with the relation between the cost of implementing a requirement to the value the requirement provides to the customer.

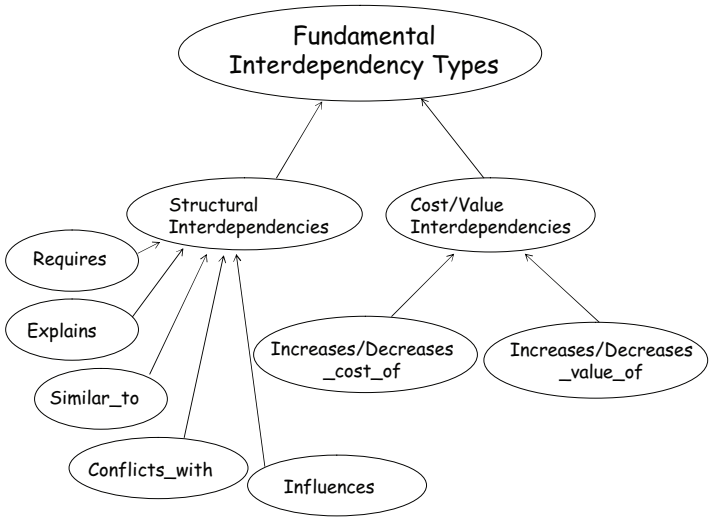

Figure 3.1: Classification of fundamental interdependency types [10]

In the paper [10] a list of activities is provided that are affected by requirements relations, such as requirements management, change management, implementation and maintenance. They recommend further research to investigate which *types* of requirements relations are critical to consider in different situations.

While Dahlstedt et al. provide a number of structural relation types (among others Requires, Explains, Conflicts_with) they do not provide formal semantics.

3.3 Requirement metamodels

A requirements metamodel is defining a requirements specification language. It captures and formally defines the concepts and relationships in the process of requirements engineering.

3.3.1 REMM - Requirements Engineering MetaModel

Vicente-Chicote et al. [40] stress the need for modeling requirements in the MDE approach. When requirements are modeled in a structured way, the reuse of requirements could be highly improved. However, there is a lack of a requirements metamodel. Furthermore they state that the requirements engineering process can only be successful when it is supported by proper tools. However, current industrial tools supporting the MDE approach leave textual requirements apart.

They propose a requirements metamodel called REMM (Requirements Engineering MetaModel). The authors note that, referring to Dahlsted et al. [10], the elements of requirement metamodels highly depend on the context it is used in. The context of the REMM metamodel is requirements reuse, although they think the concepts and relationships are applicable to a general requirements engineering approach.

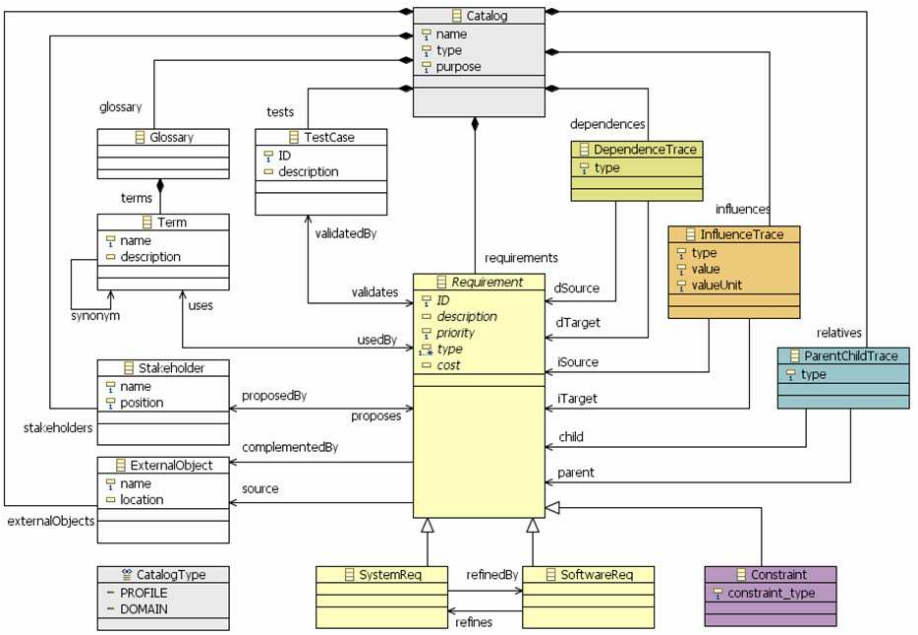

Figure 3.2: The REMM metamodel [40] (types excluded)

The reuse of requirements is supported by the metamodel (Figure 3.2) by a *Catalog*, so requirements sharing the same domain can be grouped. Requirements can be related to each other (inter-requirements traceability) or to other artifacts (extra-requirements traceability).

They developed a tool which supports the specification, validation and formatting of requirements called REMM-Studio. It is build using the Eclipse plug-in *Eclipse Modeling Framework* (EMF). REMM-Studio provides graphical editors for specifying models based on the REMM metamodel. It has a model validation tool, to be able to restrict the use of the metamodel. For example some trace relations, which are syntactically correct, make no sense. Such constraints can be specified using *Object Constraint Language* (OCL). Another capability of the tool is to generate requirements documents. It uses Model-to-Text transformations to output the modeled requirements to text files.

Vicente-Chicote et al. made a first attempt to bring the benefits of the MDE approach to the requirements engineering practice by proposing a metamodel and providing tool support for the RE process. They put constraints on the requirements models, but do not provide formal semantics for the requirements relations. Hence automated reasoning about the relations is not possible.

3.3.2 Simulation of modeled requirements

Baudry et al. [5] mention that requirements engineering encompasses a set of activities for eliciting, modeling, agreeing, communicating and validating. And that methods and tool support exists to support each activity, but they mainly remain separate. This difficults to check large requirements documents for consistency.

In the paper a metamodeling framework for requirements analysis is proposed. They implemented the metamodel to make it executable. This enables validation of the operational semantics through simulation.

Requirements are expressed as use cases and are associated with pre and post conditions. To facilitate the definition of the requirements a constrained natural language is defined. The simulation is used to detect inconsistencies and underspecification errors.

The metamodel is limited to capture only the dynamic parts of requirements, static parts are not included.

3.3.3 Metamodeling approach for requirements specification

Navarro et al. [33] note that some critical obstacles for applying RE techniques: the increasing adoption of different requirement specification approaches, the prevalence of adaptation over adoptation, and the increasing importance of the definition of domain specific languages (DSL's). Their approach is based on metamodeling to provide smooth integration and scalability of RE concepts. Navarro et al. provide guidelines to extend a core set of concepts. For defining the metamodel they studied five RE approaches (traditional, use cases, goal-oriented, aspect-oriented and variability management).

Because the five approaches use a wide diversity of terms and concepts, with many overlaps among them, they defined a core metamodel for the essential concepts. The metamodel is generic. It only considers *Artifact* and *Dependency*. The core metamodel can be customized according to the specific needs of expressiveness. The process for customizing the core is described in the paper. A case study pointed out that the proposed model was helpful for analysts, as it enables them to define and use crosscutting concerns (mainly non-functional requirements) as specific dependencies between artifacts throughout the process development.

3.3.4 Merging partial requirement specifications

Brottier et al. [6] also stress the importance of a good RE process. They note that stakeholders have different viewpoints and therefore have their requirements expressed in a different syntax. Current RE tools only support one syntax or a single input global requirement specification. Since semantic links between partial specifications remain implicit, the analysis of the requirement specifications is difficult.

The authors describe a two step process to produce a global requirements model out of a set of partial requirement specifications written in different syntaxes. This process is depicted in Figure 3.3. The first step is to parse textual requirements into an abstract syntax tree. The second step interprets the semantics of the syntax trees produced in the first step in an intermediate requirements model, and merges it into a global requirements model.

Requirements relations are not defined, and no reasoning is done on the requirements model. The detection of inconsistencies is left as future work.

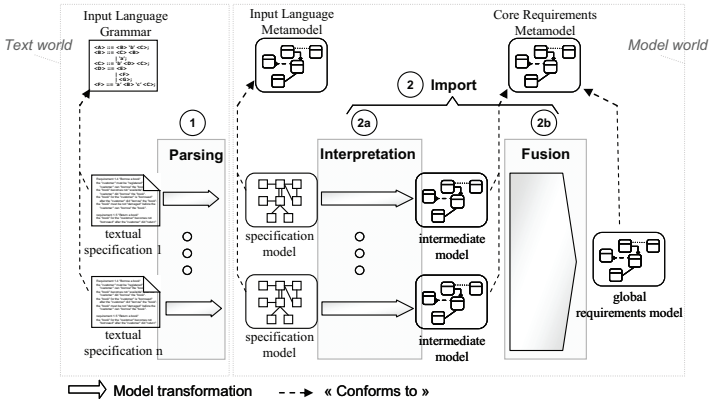

Figure 3.3: Producing a global requirements model [6]

3.3.5 Metamodels for specific domains

Some research focused at domain specific requirements modeling. In certain domains systems share a common structure. A requirements metamodel specific to the domain can help to model these systems in a structured way.

Requirements metamodel for System Families

System Families (SF's) or product lines are sets of systems that share a significant part of the development effort. Cerón et al. discuss the issue of modeling requirements in system family context [8]. Requirements must contain both common and variable parts, and also the distinction between functional and non-functional aspects have to be considered in the system family approach.

They present a metamodel which covers the requirements management needs of the system family engineering. Basic requirements relations are covered, but no formal semantics is provided.

Requirements metamodel for Web applications

Koch et al. present a metamodel for web applications [25]. The industry is using self developed approaches, but the essence of most web applications is the same. The UML-based Web Engineering metamodel (UWE) they propose is an extension of the UML metamodel.

The metamodel does not capture requirements relations.

3.4 Reasoning about requirements

Some work is done on reasoning about requirements.

Giorgini et al. [16] define a formal goal model. Goal analysis can be used the software development process during different phases. Precise formal semantics are provided for all goal relationships. In [15] a concrete example of this approach is given by adopting the goal model. Tool support is developed to enable automated reasoning. Using the tool (called T-tool) goal models can be drawn graphically. With forward and backward reasoning the user can check

whether the root goals are satisfied by the leaf goals, or which sets of leaf goals fulfill all root goals. It is merely a check for completeness, they do not check consistency. Two main limitations they describe about the approach [16] are about the definition of contribution links and the labels assignment, and that they deal with conflicts but do not resolve them.

Duffy et al. [13] provide a framework for requirements analysis using automated reasoning. The framework, called goal-structured analysis, is based on goal decomposition. The decomposition is supported by automated reasoning. No tool support is provided.

Rodrigues et al. [38] show how clustered belief revision can help to reason about evolving requirements with the presence of inconsistencies. In classical logic the presence of an inconsistency means that everything can be proven. They state this is not desirable in the context of requirements engineering were inconsistencies often arise and should be tolerated. Conflicting requirements are not always resolved during the requirements elicitation, sometimes this is postponed until the development phase The management of inconsistencies is considered important by the authors. They developed a tool that translates requirements into the disjunctive normal form and cluster prioritization.

Finkelstein et al. [14] also indicate that maintaining absolute consistency is not always possible and might constrain the development process too much. In the real-world is dealt with inconsistencies. Their approach is to formalize the handling of inconsistencies. A set of logical rules on how to act on inconsistencies is provided. They combine two lines or research: multiperspective software development in the ViewPoints framework and inconsistency handling using classical and action-based temporal logics.

3.5 Tool support

This section gives an overview of existing tool support for requirements modeling.

As source for our overview we rely on the INCOSE management tool survey [21]. Furthermore we use the preliminary results of the work of Abma [1], which focuses on traceability in requirements management tools.

We describe the traceability and reasoning capabilities of four commercial requirements management tools: IBM RequisitePro, IBM - Telelogic DOORS, Borland Caliber and Topteam Analyst.

3.5.1 IBM Rational RequisitePro

IBM RequisitePro supports two generic requirements relation types: *traceFrom* and *traceTo*. It is not possible to define custom trace types or to assign attributes or formal semantics to the relations. Besides using the transitivity of the trace relation, there is no actual reasoning done by the tool. It uses the trace types to query linked requirements. Change impact analysis is supported by visualizing the directly and indirectly related artifacts.

The tool does not provide a consistency checking mechanism.

3.5.2 IBM - Telelogic DOORS

IBM - Telelogic DOORS allows custom relation types. The types are defined in separate modules. A number of attributes can be assigned and the user can constrain the type of the requirements linked with a specific relation type (i.e. only allow nonfunctional requirements). The semantics of the relation types are not used during analysis. The transitive property of the relation is used to query for indirect relations. There is a basic support for change impact analysis. Traces can be followed up or down to analyze possible impacted requirements.

The tool does not support consistency checking of relations.

3.5.3 Borland Caliber

Like RequisitePro there is only one relation type in Borland Caliber. It is not possible to specify custom relation types or assign attributes to relations. The transitive property of the relation is used to query for indirect relations. Borland Caliber does not support change impact analysis.

The tool does not provide a consistency checking mechanism.

3.5.4 Topteam Analyst

Topteam Analyst supports thee traceability link types: *traces into*, *impacts* and *used in*. It is not possible to specify custom relation types or assign attributes to relations. The support for change impact analysis is very limited. No reasoning is done hence no indirect relations are visible to the user.

Consistency checking is not supported by Topteam Analyst.

3.6 Conclusion

In this chapter we described related work. In the first section we describe the current work done on requirements relations. Dahlstedt et al. [10] notes the importance of requirements relations to make sound decisions during the software development process. They provide an overview of relation types used in literature and classified these in two categories. They conclude that more research is needed to identify which relation types are important in different situations.

In Section 3.3 we described research done on requirements metamodels. The research is motivated by the lack of structure in requirements documents. Metamodels can help to provide a common structure. Because there are many requirements engineering approaches using different concepts, the metamodels are either very generic, or very specific to a RE approach or domain.

Section 3.4 describes the work done on requirements reasoning. Many of the works are targeting the goal-oriented requirements engineering approaches. Rodrigues et al. [38] and Finkelstein et al. [14] note that inconsistencies between requirements should be tolerated. In both works is mentioned that inconsistencies do not have to be resolved during the requirements elicitation, this resolving may be postponed to later stages of the development.

In Section 3.5 we describe the traceability and reasoning capabilities of commercially available requirements management tools. We conclude that they do

not support reasoning about requirements. Only the transitivity of a trace relation is used by some tools to query for indirectly related requirements. Some of the tools allow the user to define custom relations, but it is not possible to define semantics. Consistency checking of requirements models is not performed by the tools we investigated.

In the next chapter we describe the approach of Göknil et al. [19]. A core requirements metamodel is described and the requirements relations in the model are provided with formal semantics. This enables reasoning about requirements relations and consistency checking. The metamodel of Göknil et al and the formalization form the base of our work.

4

Formalization of Requirements and Relations

4.1 Introduction

The main goal of this work is to develop tool support for reasoning about formalized requirement relations [19][17]. In this chapter we explain the formalization of requirements and requirement relations and reasoning.

In Section 4.2 we describe the requirements metamodel. Section 4.3 describes the formalization of requirements. Section 4.4 describes the formalization of requirements relations. In Section 4.5 we describe how requirements relations are mapped to their formal definitions. In Section 4.6 we describe how the formalization of requirements relations can be used to infer implicit relations and to check for consistency. Section 4.7 summarizes this chapter.

4.2 Requirements metamodel

Several requirements engineering approaches exist: goal-oriented [32], aspect-driven [37], variability management [31], use-case [9], domain-specific [35][25], and reuse driven [28]. All these approaches use different methods to model requirements.

From these approaches Göknil et al. [19] derived a requirements metamodel by taking common entities. The essence of the metamodel is shown in Figure 4.1. For simplicity we left out entities which are not used for the tool support such as Stakeholder, TestCase and AdditionalDescription.

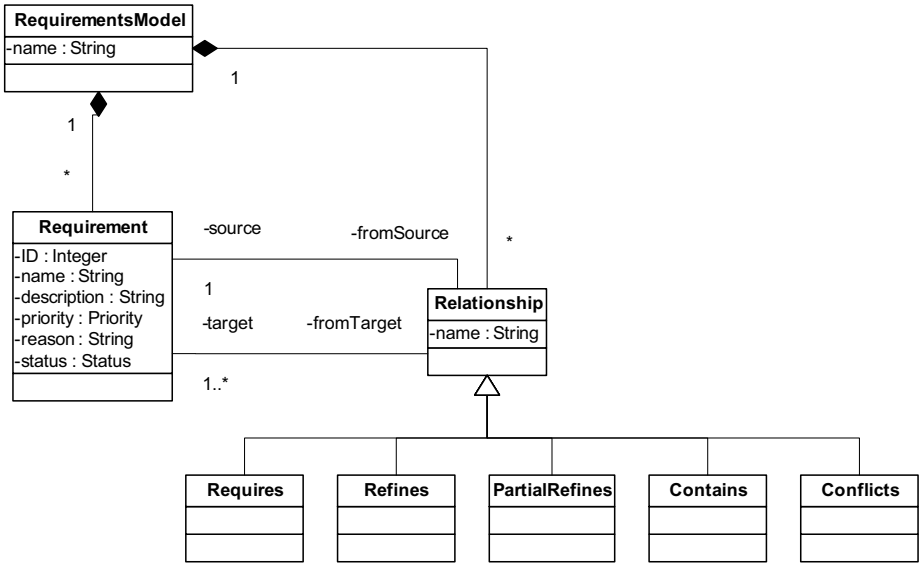

Figure 4.1: The core requirements metamodel [19]

The metamodel has a main entity `RequirementsModel`. Each requirements model has zero or more requirements. A `Requirement` has a number of attributes such as a name, description and a status. Requirements can be related with each other through a `Relationship`. The metamodel denotes five relationship types: `Requires`, `Refines`, `PartialRefines`, `Contains` and `Conflicts`.

According to [17] these relations have the following informal definition in literature:

requires A requirement R_1 *requires* a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled.

refines A requirement R_1 *refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to it.

partial-refines A requirement R_1 *partial-refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to parts of R_2 and excluding the unrefined parts of R_2.

contains A requirement R_1 *contains* requirements $R_2 \dots R_n$ if $R_2 \dots R_n$ are parts of the whole R_1 (part-whole hierarchy).

conflicts A requirement R_1 *conflicts* with a requirement R_2 if the fulfillment of R_1 excludes the fulfillment of R_2 and vice versa.

These definitions are informal. The requirements relations are formalized in first-order logic. In Section 4.4 we describe their formalization.

4.3 Formalization of requirements

A requirement is assumed being "a property which must be exhibited by a system" [2]. A requirement R is defined as a tuple $\langle P, S \rangle$ where P is the property

and S is the set of systems that satisfy P (i.e. $\forall s \in S : P(s)$) [19].

Property P can be represented in a conjunctive normal form (CNF) as follows: P = $(p_1 \wedge \ldots \wedge p_n)$; where n ≥ 1 and p_n is the disjunction of literals.

4.4 Formalization of requirements relations

The requirements relations defined in the metamodel are formalized in first-order logic. In this section we provide the formalizations.

4.4.1 Formalization of requires

Let $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$ be requirements.

- R_1 *requires* R_2 iff $\forall s \in S_1 : s \in S_2$ and $\exists s \in S_2 : s \notin S_1$

From the above definition is concluded that $S_1 \subset S_2$. This subset relation between sets of systems defines the *requires* relation as *non-reflexive*, *non-symmetric* and *transitive*.

4.4.2 Formalization of refines

Let $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$ be requirements. P_1 and P_2 are formulas and the conjunctive normal form of P_2 is:

- $P_2 = (p_1 \ldots p_n) \wedge (q_1 \ldots q_m)$; $n \geq 1$, $m \geq 0$

Let $p'_1, p'_2, \ldots, p'_{n-1}, p'_n$ be the disjunction of literals such that $p'_j \rightarrow p_j$ for $j \in 1..n$.

- R_1 *refines* R_2 iff P_1 is derived from P_2 by replacing every p_j in P_2 with p'_j for $j \in 1..n$ such that the following two statements hold:

  $P_1 = (p'_1 \ldots p'_n) \wedge (q_1 \ldots q_m)$; $n \geq 1$, $m \geq 0$
  $\exists s \in S_2 : s \notin S_1$

From the definition is concluded that if P_1 holds for a given system s then P_2 also holds for s ($\forall s \in S_1 : s \in S_2$). Based on $\exists s \in S_2 : s \notin S_1$ and $\forall s \in S_1 : s \in S_2$, a subset relation between the satisfying sets of systems is concluded: $S_1 \subset S_2$.

The *refines* relation is *non-reflexive*, *non-symmetric* and *transitive*. If R_1 *refines* R_2 then, obviously, R_1 *requires* R_2.

4.4.3 Formalization of partial-refines

Let $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$ be requirements. P_1 and P_2 are formulas and the conjunctive normal form of P_2 is:

- $P_2 = (p_1 \ldots p_n) \wedge (q_1 \ldots q_m)$; $n, m \geq 1$

Let $q'_1, q'_2, \ldots, q'_{m-1}, q'_m$ be the disjunction of literals such that $q'_i \rightarrow q_i$ for $i \in 1..m$.

- R_1 *partial-refines* R_2 iff P_1 is derived from P_2 by replacing every q_i in P_2 with q'_i for $i \in 1..m$ and excluding others (p_i for $i \in 1..n$) such that the following two statements hold:

  $P_1 = (q'_1 \dots q'_m)$

  $\exists s \in S_2 : s \notin S_1$, $\exists s \in S_1 : s \notin S_2$, and $\exists s \in (S_1 \cap S_2)$

The *partial-refines* relation is *non-reflexive*, *non-symmetric* and *transitive*.

Decomposition of partial-refines

The *partial-refines* relation is a special combination of the *refines* and *contains* relation. The *partial-refines* can be decomposed using a temporal requirement ($R_{T12} = \langle S_{T12}, P_{T12} \rangle$):

- contains(R_2, R_{T12}) $\wedge$ refines(R_1, R_{T12}) iff partial-refines(R_1, R_2)
- refines(R_{T12}, R_2) $\wedge$ contains(R_{T12}, R_1) iff partial-refines(R_1, R_2)

Note that this mapping between the *contains* and *refines* and the partial-refines uses the special temporal requirement R_{T12}. A *contains* and *refines* between normal requirements does not map to a *partial-refines*.

4.4.4 Formalization of contains

Let $R_1 = \langle P_1, S_1 \rangle, R_2 = \langle P_2, S_2 \rangle, \dots, R_k = \langle P_k, S_k \rangle$ be requirements where $k \geq 2$. $P_2, P_3, \dots, P_k$ are formulas in conjunctive normal form:

- $P_i = (p'_1 \dots p'_{m_i})$; $m_i \geq 1, i \in 2..k$
- R_1 *contains* $R_2, \dots, R_k$ iff P_1 is derived from $P_2, P_3, \dots, P_k$ as follows:

  $P_1 = P_2 \wedge P_3 \wedge \dots \wedge P_k \wedge P'$
  where P' denotes properties that are not captured in $P_2, P_3, \dots, P_k$

Completeness of the decomposition is not assumed. From the definition is concluded that if P_1 holds then $P_2, P_3, \dots, P_k$ also hold, and if $P_2, P_3, \dots, P_k$ hold then P_1 does not have to hold. Hence is concluded that $S_1 \subset S_2, S_1 \subset S_3, \dots,$ and $S_1 \subset S_k$.

The *contains* relation is *non-reflexive*, *non-symmetric* and *transitive*.

4.4.5 Formalization of conflicts

Let $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$ be requirements.

- R_1 *conflicts* with R_2 iff $\neg \exists s : (s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s))$

There are several inconsistency types which could be seen as a conflict between two requirements. With *conflicts* is meant: excluding the fulfillment of requirements types [39]. If R_1 *conflicts* R_2 then there does not exist a system that satisfies both.

4.5 Mapping of requirements relations to formal definitions

In the previous section we described how the requirements relations are formalized in first-order logic. We summarize the formal definition to illustrate the mapping between requirements relations and formal definitions. These mappings are used to infer implicit relations and to check for consistency.

We distinguish two types of mappings of requirements relations to formal definitions: a mapping to satisfying sets of systems, and a mapping to formula relations between properties of requirements.

4.5.1 Mapping to sets of satisfying systems

From the formalization of the requirements relations we can create a mapping between requirements relations and relations between sets of systems. Table 4.1 gives this mapping. The direction of the mapping is denoted with arrows.

Table 4.1: Mapping of relation types to systems relation

Relation type	Mapping direction	Systems relation[1]
R_1 requires R_2	$\Longleftrightarrow$	$S_1 \subset S_2$
R_1 refines R_2	$\Longrightarrow$	$S_1 \subset S_2$
R_1 contains R_2	$\Longrightarrow$	$S_1 \subset S_2$
R_1 conflicts R_2	$\Longleftrightarrow$	$S_1 \cap S_2 = \varnothing$
R_1 partial-refines R_2		*See Section 4.4.3*

[1] S_1 and S_2 are the sets of systems satisfying R_1 and R_2 respectively

A *requires* relation can be derived from a *refines* or a *contains* relation by using this mapping. The disjointness of systems ($S_1 \cap S_2 = \varnothing$) is concluded on basis of the definition of the *conflicts* relation (Section 4.4.5).

The *partial-refines* relation does not have a mapping to either a subset relation or a disjointness of sets of systems. As mentioned in Section 4.4.3 it is decomposed into a *refines* and *contains* relation. Therefore the *partial-refines* indirectly maps to the sets of satisfying systems.

4.5.2 Mapping to formula relations

In [17] the relations between properties are defined. In this work we refer to them as *formula relations*. The properties are the properties (P) of requirements ($R = \langle P, S \rangle$).

Each of the formula relations has a different formalization. The details of the formalization of formula relations are out of the scope of this work. The formalization itself is not needed for reasoning on requirements.

Table 4.2 provides an overview of the formula relations and their logical property characteristics. All formula relations are *non-reflexive*, *non-symmetric* and, except for all-equals-in, *transitive*.

Table 4.2: Logical property characteristics of the formula relations

formula relation	reflexive	symmetric	transitive
all-in-whole	no	no	yes
all-in-part	no	no	yes
all-implies-in	no	no	yes
some-implies-in	no	no	yes
all-equals-in	no	no	no

Some pairs of formula relations are defined as disjoint:

- all-in-part and all-in-whole are disjoint
- all-equals-in and some-implies-in are disjoint
- all-equals-in and all-implies-in are disjoint

This means that there is a contradiction if two disjoint relations exist between the same two requirement properties.

Some combinations of formula relations imply the existence of another formula relation. A number of **inference rules for formula relations** is provided:

all-in-part$(P_1, P_2) \wedge$ all-in-whole(P_2, P_3)	$\Rightarrow$	all-in-part(P_1, P_3)
all-in-whole$(P_1, P_2) \wedge$ all-in-part(P_2, P_3)	$\Rightarrow$	all-in-part(P_1, P_3)
some-implies-in$(P_1, P_2) \wedge$ all-implies-in(P_2, P_3)	$\Rightarrow$	all-implies-in(P_1, P_3)
all-implies-in$(P_1, P_2) \wedge$ some-implies-in(P_2, P_3)	$\Rightarrow$	all-implies-in(P_1, P_3)
some-implies-in$(P_1, P_2) \wedge$ all-equals-in(P_2, P_3)	$\Rightarrow$	some-implies-in(P_1, P_3)
all-implies-in$(P_1, P_2) \wedge$ all-equals-in(P_2, P_3)	$\Rightarrow$	all-implies-in(P_1, P_3)
all-equals-in$(P_1, P_2) \wedge$ all-implies-in(P_2, P_3)	$\Rightarrow$	all-implies-in(P_1, P_3)

Table 4.3 provides the mapping of requirements relations to formula relations. The mapping is in both directions. The *requires* and *conflicts* relation do not map to a formula relation.

Table 4.3: Mapping of requirement relations to formula relations

Relation type	Formula relation[1]
R_1 requires R_2	-
R_1 refines R_2	all-in-whole$(P_1, P_2) \wedge$ some-implies-in(P_1, P_2)
R_1 contains R_2	all-in-part$(P_2, P_1) \wedge$ all-equals-in(P_2, P_1)
R_1 conflicts R_2	-
R_1 partial-refines R_2	all-in-part$(P_1, P_2) \wedge$ all-implies-in(P_1, P_2)

[1] P_1 and P_2 are the properties of requirement R_1 and R_2 respectively

4.6 Inferencing and Consistency Checking

With the formalization of requirements and requirements relations we are able to infer implicit relations and to check the modeled requirements for consistency. In the section we provide some examples to illustrate how this is done.

4.6.1 Inference of implicit requirements relations

Suppose we have three requirements R_1, R_2 and R_3 which are related as follows:

$(R_1$ *refines* $R_2) \wedge (R_2$ *contains* $R_3)$.

Using the formalization we can infer that (R_1 *requires* R_3).
We use the mapping to sets of systems (see Section 4.5.1). The *refines* and *contains* relations both map to a subset relation. This results in: $S_1 \subset S_2$ and $S_2 \subset S_3$. A subset relation between two sets of systems can be mapped to a *requires* relation. So we conclude:

$(R_1$ *requires* $R_2) \wedge (R_2$ *requires* $R_3)$.

By using the transitivity of the *requires* relation we can infer that (R_1 *requires* R_3).

4.6.2 Consistency Checking

A requirements model is considered consistent when there are no contradicting requirements relations. The requirements relations are formalized to both sets of satisfying systems and properties in conjunctive normal form. For the inference of implicit requirements relations the logical property characteristics of the relationship types combined with the mapping to relations between sets of satisfying systems suffices. To detect all possible contradictions, however, we also need the formalization of the formula relations.

We give an example showing the need for the formula relations in addition to the sets of systems.

Suppose we have modeled two requirements R_1 and R_2 as follows:

$(R_1$ *refines* $R_2) \wedge (R_1$ *contains* $R_2)$.

The mapping to sets of systems will not cause a contradiction, both relations map to $S_1 \subset S_2$. However, when we map the relations to formula relations we get this:

$(R_1$ *all-in-whole* $R_2) \wedge (R_1$ *some-implies-in* $R_2)$.

and

$(R_2$ *all-in-part* $R_1) \wedge (R_2$ *all-equals-in* $R_1)$.

Because the *all-in-whole* and *all-in-part* relations are disjoint, there is a contradiction between the formula relations of R_1 and R_2. Also, the (symmetric) *all-equals-in* relation is disjoint with *all-equals-in*.

Therefore we can conclude that (R_1 *refines* R_2) and (R_1 *contains* R_2) are contradicting.

4.7 Conclusion

In this chapter we described the formalization of requirements and requirements relations contained in the core requirements metamodel. The formalization and

the core metamodel are a result of research by Göknil et al. [19][18][17].

The formalization of requirements relations creates a mapping to sets of systems and to formula relations. These mappings together with the logical property characteristics of the relations can be used to *infer* implicit requirements relations.

The formalization puts constraints on modeled requirements relations. Set theoretic constraints apply to the sets of systems satisfying requirements. For example, a set cannot be a subset of another set while simultaneously being disjoint with that set. For the formula relations additional constraints are provided. These constraints enable automatic consistency checking.

In the next chapter we describe the requirements for a tool supporting the modeling of requirements conforming to the metamodel we described in this chapter. The tool should also be capable to infer implicit relations and check for consistency by applying the formalization discussed in this chapter.

5 Requirements for TRIC

5.1 Introduction

This chapter describes the specific requirements for the tool. The tool is named **TRIC**: Tool for Requirement Inference and Consistency checking. We start with defining the purpose and scope of this tool, followed by a list of functional and non-functional requirements. During the development of the tool, the requirements have slightly changed. In this chapter we describe the current state of the requirements. In Chapter 8 we evaluate to what extend the requirements are met by our implementation.

Section 5.2 describes the scope and purpose of the tool. Section 5.3 lists the stakeholders of the tool. In Section 5.4 the requirements are listed. Section 5.5 gives the planned releases (iterations) of the tool. Section 5.6 summarizes this chapter.

5.2 Scope and purpose

The tool will act as a proof-of-concept of the requirements metamodeling approach. Goal of the tool is to infer implicit requirement relations, and to detect inconsistent requirement relations. This is done by creating an environment where requirements can be entered into a model. These models conform to the *requirements metamodel*. The metamodeling approach intends to customize this metamodel to fit into an existing requirements engineering approach. Models can be stored to and retrieved from the file system.

Once the requirements and their relations are modeled, a reasoning engine can infer implicit relations between requirements, using metamodel specific inference rules. Comparable to the inferencing of implicit relations, a consistency checking engine detects inconsistent relations.

To support analysis of inferred relations and inconsistencies, a visualization engine provides a graphical explanation of an inferred relation.

Due to the proof-of-concept nature of the tool, the focus is on the reasoning on requirements. Providing a user-friendly requirements management tool is not our goal. As we will mention later on, interoperability with existing tools could alleviate the work needed to conduct a (large) case study.

5.3 Stakeholders

The metamodeling approach distinguishes two types of stakeholders, which are listed below.

Metamodel engineer Is responsible for customizing the core requirements metamodel, to fit the models to the used requirements engineering approach. Also needs to specify the inference and consistency rules specific to the customized metamodel.

Requirements engineer Is responsible for entering the requirements, and relating them using the available relation types in the used metamodel. The requirements engineer uses the tool to infer implicit relations, check the model for consistency, and to analyze the results of the inference and consistency checker engine.

The tasks of the metamodel engineer are not supported by the tool. The requirement metamodel should be specified manually, or by using an external tool.

5.4 Requirements

We have the following requirements for the tool and separated functional and non-functional requirements.

5.4.1 Functional requirements

T1 Modeling Environment
The tool shall support a modeling environment for requirements models. The requirements models must conform to the *core requirements metamodel* or to a customization of this metamodel.

T2 Modeling Requirements
The tool shall allow requirements engineers to enter requirements into a model.

T3 Inferencing
The tool shall provide an inference engine that will support the inference of implicit requirements relations.

T4 Consistency Checking
The tool shall provide a consistency checking engine supporting the detection of inconsistencies.

T5 Custom inference rules
The tool shall allow metamodel engineers to specify inference rules for (customizations of) the *core requirements metamodel*, based on the formalization of requirements relations.

T6 Custom consistency rules
The tool shall allow metamodel engineers to specify consistency rules for a specific customization (for instance, a requirement cannot be related to itself).

T7 Detecting inconsistencies
The tool shall report detected inconsistencies to the user.

T8 Derivation analysis
The tool shall support analysis of reasoner results by means of a visualization engine.

5.4.2 Non-functional requirements

T9 Extendibility
The tool shall be extendible to support future research on requirements modeling. For instance, on change impact analysis.

T10 Scalability
The time behavior of the prototype is not an issue, although the tool shall support models large enough to conduct case studies.

T11 User-friendliness
The tool is required to support the exploration of case studies by researchers.

T12 Interoperability
The tool should use an open standard for storage of the requirements models and metamodels to enable the import and export of other tool's models.

5.4.3 Constraints

In order to limit the size of the project, we have the following constraints:

1. The *core requirements metamodel* is fixed and cannot be changed during operation.
2. Once models are entered, the corresponding metamodel cannot be changed.
3. The tool should be developed as a single Eclipse RCP application.

5.5 Planned releases

Before starting the implementation, we planned a couple of releases. We did this to support an iterative approach. This enabled us to verify partial functionality of the system while extending the tool at the same time.

First release
Initial version, linking the most important components together.

1. Core requirements metamodel in modeling environment.
2. a simple (non-graphical) modeling environment for requirements
3. an inference engine to reason on the formalization of requirements relations

Second release
Version of the tool which is 'ready enough' for a case study.

4. requirements modeling environment which supports the entering of requirements and assigning relations
5. a consistency checking engine to detect inconsistencies of requirements relations

Third release
Final version, adding visualization to make the tool more user-friendly.

6. provide visualization to support reasoner analysis

5.6 Conclusion

In this chapter we listed the requirements for the tool support. The purpose of the tool is to support the requirements metamodeling approach and to provide automatic reasoning on and consistency checking of modeled requirements using the formalizations described in Chapter 4.

Two stakeholders are identified: metamodel engineers and requirement engineers. A number of constraints is defined to limit the size of the project. Three releases are planned to support an iterative development approach for the tool.

In the next chapter (Chapter 6) we take this requirements as a base for the design of the tool. At the evaluation (Chapter 8) we use the requirements to evaluate the implementation of the tool.

6 Design of TRIC

6.1 Introduction

In the previous chapter we enumerated the requirements and gave an overview of the functionality of the tool we developed. This chapter covers the internal design of the tool.

In Section 6.2 we describe the high level design of TRIC. In Section 6.3 we describe the activities supported by the tool. In Section 6.4 we conclude the chapter.

6.2 High level design

We used the requirements for the tool (described in Chapter 5) to identify a number of components. These components constitute the core of TRIC.

- a Modeling Environment
- an Inference Engine
- a Consistency Checking Engine
- a Visualization Engine

In the architectural design we show how these components are connected with each other. In the subsequent sections of this chapter we provide a detailed description of each component.

We chose to develop the tool using the Eclipse Rich Client Platform (RCP). This reduced the number of design alternatives for the user-interface. Furthermore we prefer to use existing components and technologies rather than to develop common components (such as user-interface components, first-order logic reasoners) ourselves.

TRIC is an interactive application. Therefore we apply the Model-View-Controller (MVC) architectural pattern (which is discussed by Buschmann et al. in detail [7]). Views (i.e. the user interface) are separated from the data they display. The data is contained in models.

One of the main benefits of the MVC pattern is the ability to have multiple views of the same model. Another benefit is the synchronization of views and models (i.e. the user-interface is displaying what is currently in the model).

We use the Eclipse RCP as a platform for the views and use its builtin mechanisms to forward events to controllers.

The requirements models are represented as Web Ontology Language (OWL) ontologies. In Section 7.3 we explain the details of creating and updating requirements models.

To separate data from application logic, we applied the façade pattern [27]. By using Data Access Objects, the application is able to read and manipulate models, without any dependency to data format. The Data Access Objects also simplify the code needed to interact with the requirements models.

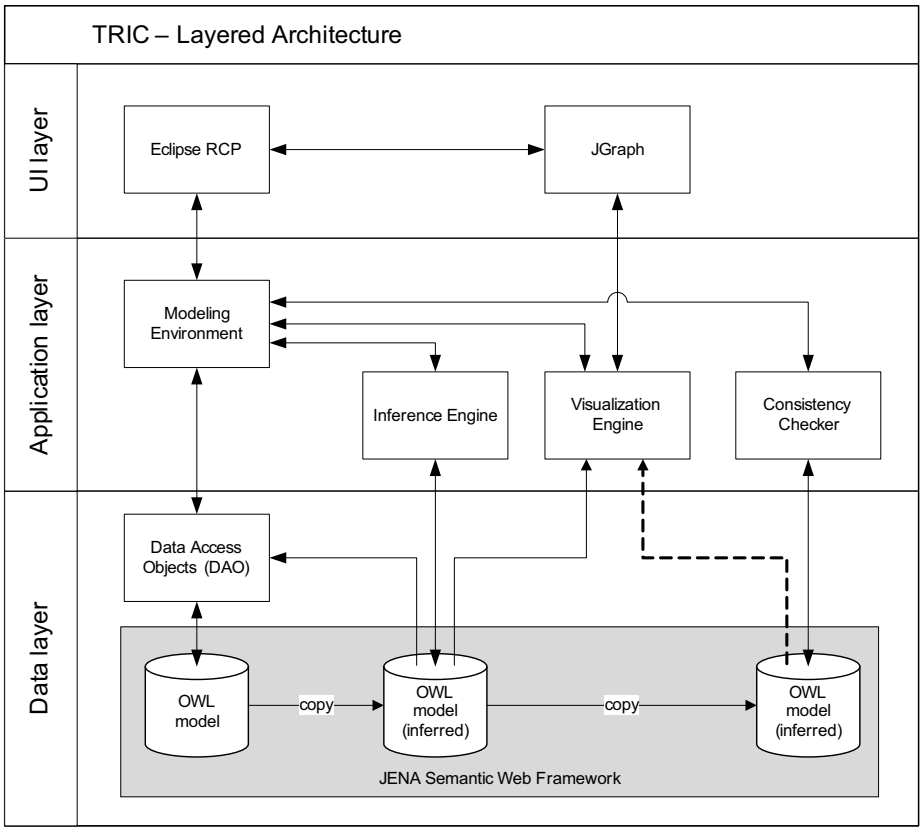

Figure 6.1: Layered architecture of TRIC

Figure 6.1 shows the architectural design of the tool. We distinguish three layers: a data layer containing the model data, a user interface layer enabling user interactivity, and an application layer in-between.

The core of the tool is the application layer. The Modeling Environment is the bridge between the user-interface and the requirements model. Reasoning on requirements models (inferencing and consistency checking) is done straight on the OWL models (without Data Access Objects in-between). To prevent pollution of the requirements model with inferred relations and other elements added, we reason on *copies* of the original model. We will come back on this in the section about the Inference Engine (Section 7.4). The arrow between the output model of the Consistency Checking Engine and the Visualization Engine is dashed because the current implementation does not visualize inconsistencies.

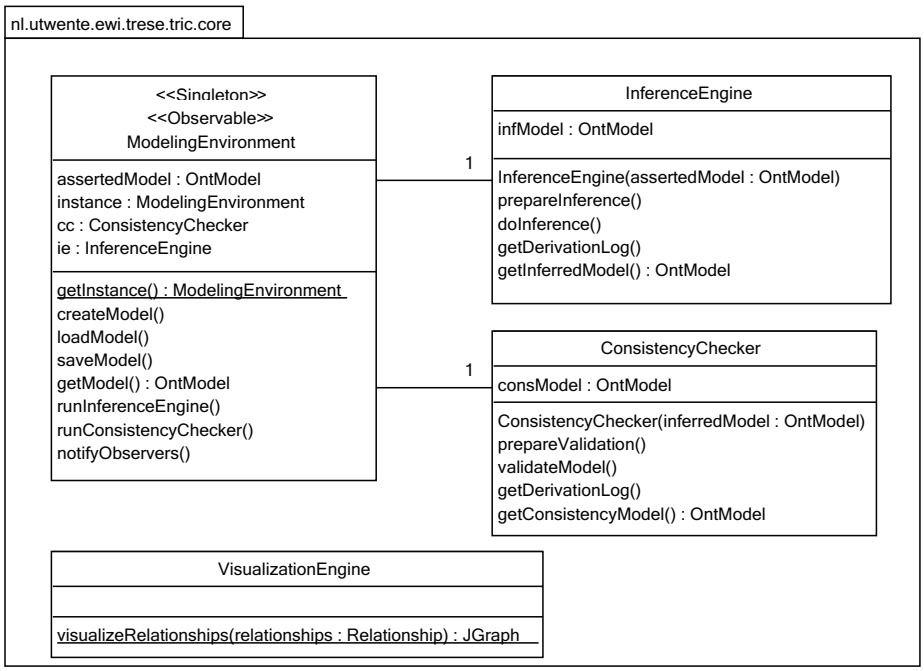

Figure 6.2: Package diagram of the core package

The package diagram of the core package is given in Figure 6.2. The classes in the Application Layer are considered as core. The Singleton-pattern is applied to the ModelingEnvironment class. This simplifies the access to the modeling environment from other classes. By calling `ModelingEnvironment->getInstance()` the ModelingEnvironment instance can be obtained. The ModelingEnvironment has one instance of the InferenceEngine and one instance of the ConsistencyChecker. The VisualizationEngine class is used statically and therefore not directly connected to other classes.

When a new requirements model is created, the core requirements metamodel is automatically imported. This metamodel is stored in a separate ontology. In Section 7.3 about the Modeling Environment we will elaborate on this.

We summarize the responsibilities of each component as follows:

- **Modeling environment** (Section 7.3)
  - taking care of creation, storage and retrieval of requirements models,
  - guarantying conformance of a model to the requirements metamodel.
- **User Interface**
  - provide a simple form-based editor to enter and modify requirements,
  - provide a method to assign relations between requirements,
  - provide a view on the requirements in the model,
  - be able to control the services provided by the system (run the Inference Engine, run the Consistency Checker, etc.).
- **Inference Engine** (Section 7.4)
  - infer all implicit relations between requirements in a given model (even if they do not make sense), by using metamodel specific inference rules,
  - keep track of given and inferred relations,
  - provide a method to explain a derivation.
- **Consistency Checker** (Section 7.5)
  - check for consistency, by using (metamodel specific) validation rules,
  - provide a list of found inconsistencies.
- **Visualization Engine** (Section 7.6)
  - provide a compact visual explanation of an inferred relation.

6.3 Activities supported by the tool

In the previous section we described the high-level design. This section describes the activities supported by the tool. In Section 5.3 we listed the two stakeholders of the tool: metamodel engineers and requirements engineers.

The activities of the metamodel engineer (specifying the requirements metamodel and specification of inference and consistency checking rules) are not supported by the tool. The metamodel and the reasoner rules are stored in separate files and can be modified using other tools. The metamodel is an OWL ontology and can be created using an OWL editor. We used Protégé [36] to create the metamodel.

Figure 6.3 gives the activity diagram of the modeling process. There is a division between the activities of the metamodel engineer (which are not supported by TRIC) and the activities of the requirements engineer. In the diagram the activities of the requirements engineer are all supported by the tool. The *decisions* depicted in the diagram are made by the user, not by the tool.

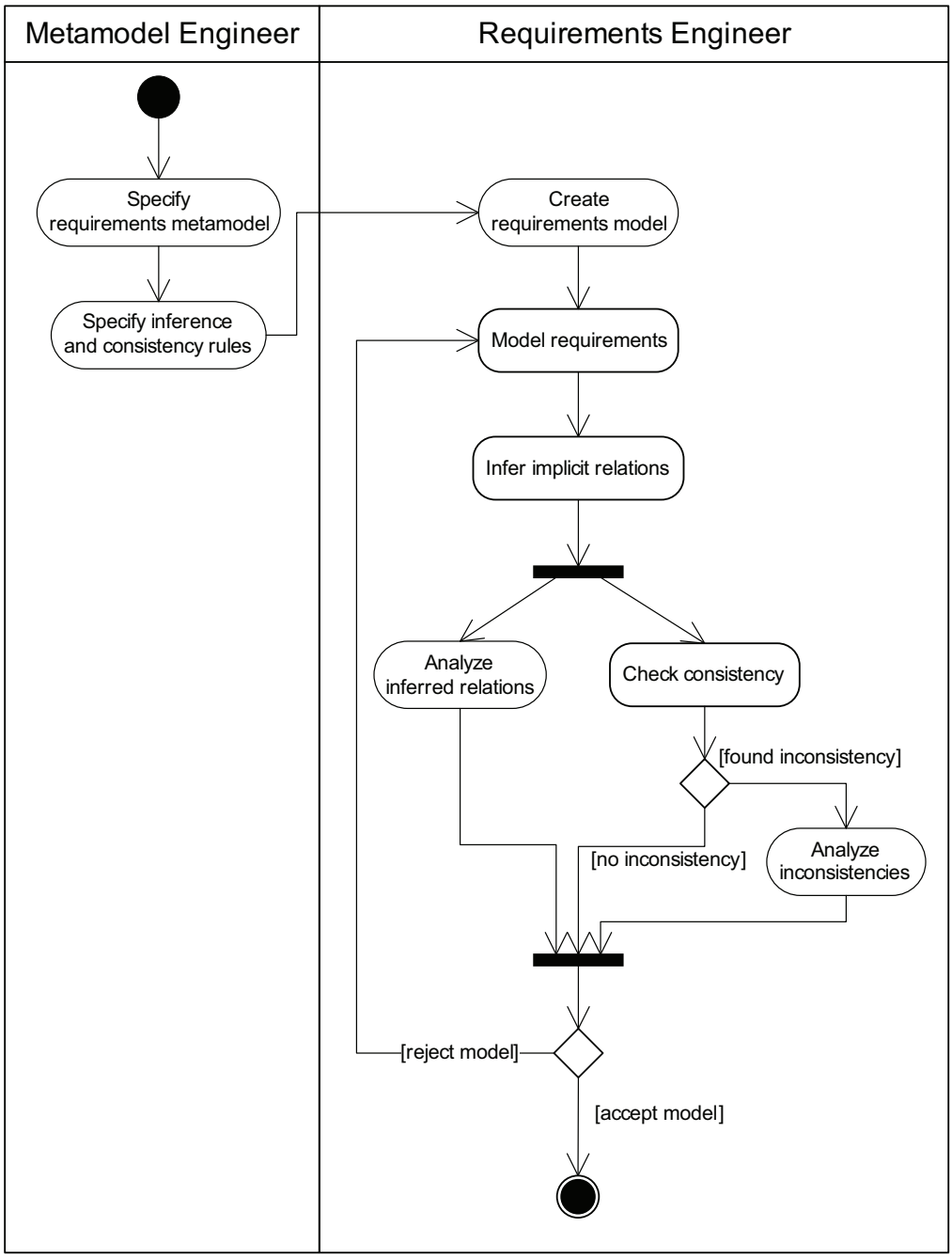

Figure 6.3: Activity diagram of the modeling process

For the requirements engineer the following activities are supported:

- **Create requirements model**
  Create a new requirements model. This model conforms to the requirements metamodel specified by the metamodel engineer.
- **Model requirements**
  Add new, or update existing requirements in the model. Relating requirements using the relationship types specified in the metamodel is also part of the modeling activity.
- **Infer implicit relations**
  The requirements engineer can manually run the inference engine. The inference rules specific to the used metamodel are used to reason on the modeled requirements. As a result, implicit requirement relations are inferred. These implicit relations become visible in the user-interface.
- **Analyze inferred relations**
  The requirements engineer can inspect the inferred relations by browsing the modeled requirements. The implicit relations must match the related requirements. When an inferred relation does not match, a given relation is applied incorrectly.
  To detect which given relations are used to infer a certain inferred relation the Visualization Engine can be used. This visualizes the path followed by the inference engine. As an alternative a textual derivation log can be retrieved.
  Parallel to the analysis of the inferred relations, the requirements engineer can check the model for consistency.
- **Check consistency**
  The entered requirements model can be checked for consistency using the metamodel specific consistency rules. These consistency rules are specified by the metamodel engineer.
  As depicted in Figure 6.3, the inference of implicit relations is done *prior* to consistency checking. We explain this sequence in the subsection hereafter.
  When the Consistency Checking Engine is finished, the requirements engineer is prompted with a dialog box. Either zero, or one or more inconsistencies are found. When one or more inconsistencies are found, the requirements engineer can decide to analyze them.
- **Analyze inconsistencies**
  A textual description is written to the console when inconsistencies are encountered by the Consistency Checking Engine. This output can be used to detect which requirements are involved in an inconsistency. The current implementation of TRIC does not provide a visualization of the requirements causing an inconsistency.
- **Iterative modeling**
  After the inference of implicit relations and possible consistency checking of the model, it is up to the requirements engineer to decide whether to accept or reject the requirements model. The requirements model can be updated at any time and the inference and consistency checking activities can be repeated until the requirements engineer is satisfied.

It is important to notice that consistency checking is only possible *after* inferring the implicit relations. Besides a technical reason which we explain in Section 7.4, there is also a general rationale. After checking for consistency, a requirements engineer expects the tool to report whether the model is consistent. On the basis of the given relations only, the system might conclude consistency while the inferred model is inconsistent. Because this can be misleading, we always want to infer all implicit relations before checking the model's consistency.

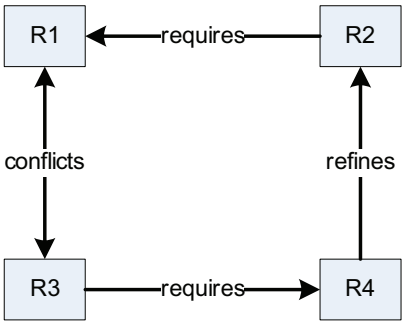

Figure 6.4: Example of an inconsistency undetectable without inference

In Figure 6.4 an example is shown. Without inferring the implicit relations, the tool cannot conclude there is an inconsistency. However, R2, R3 and R4 both require and conflict with R1.

The activity diagram does not show activities related to opening and storing models. The tool allows the requirements engineer to store the model to the file system at any time, so the iterative modeling process can be continued at another moment.

In the next sections we describe the internal design of each component in more detail.

6.4 Conclusion

In this chapter we described the design of the tool. We chose for a layered architecture to separate data from application and presentation code. By applying the Model-View-Controller (MVC) architectural pattern we ensure a synchronized view of the model.

We identified components of the tool and described their responsibilities. An activity diagram of the modeling process is provided and discussed. We chose to infer implicit relations prior to checking for consistency, because without inferencing a check for consistency may lead to a false positive.

In the next chapter we describe the implementation of TRIC.

7 Implementation of TRIC

7.1 Introduction

This chapter describes the implementation of TRIC. We describe the implementation of each component (identified in Chapter 6) in separate sections. We start with describing how we mapped the formalization of requirements and requirements relations (Chapter 4) to OWL.

In Section 7.2 we describe how we use OWL to reason about requirements. Section 7.3 describes the Modeling Environment. In Section 7.4 we describe the implementation of the Inference Engine. Section 7.5 describes the Consistency Checking Engine. Section 7.6 describes the Visualization Engine. Section 7.7 demonstrates the usage of the tool and gives some screenshots of the graphical user-interface. Section 7.8 summarizes this chapter.

7.2 Usage of OWL to reason about requirements

Reasoning on requirements models using first-order logic is the main purpose of the tool. Within the Eclipse project there is a Eclipse Modeling Framework (EMF) . This framework is frequently used in Model Driven Architecture tool prototypes. However, to the best of our knowledge, there exist no first-order logic reasoners which are able to reason on EMF models. Since OWL offers both a structured representation of (meta)models and a lot of first-order logic reasoners for OWL exist, we express requirements (meta)models as OWL ontologies.

7.2.1 Requirements metamodel as OWL ontology

The requirements metamodel, as discussed in Chapter 4, is expressed in UML . Because TRIC aims at modeling requirements and their relations, we took

the following elements of the metamodel: `Requirement`, `Relationship` and subclasses of `Relationship`.

The Relationship class is an association class, relating two requirements in a specific direction. While it is possible to retain this association class in OWL, we chose to replace the class with an association. Associations are expressed in OWL as *Object Properties*, and can be used directly in reasoning. And contrary to UML, these associations (Object Properties) can have logical property characteristics (such as symmetry and transitivity). This is very useful since these logical properties are essential for reasoning.

7.2.2 Handling of metamodels and models

When a model is created by the Modeling Environment it will add the ontology with the metamodel as an import (with prefix `mm:`). Requirements are created as RDF resources, being individuals of the Requirement class in the metamodel. Figure 7.1 gives an example of the RDF notation of a minimal model where R2 refines R1, the details of Requirement 2 are omitted.

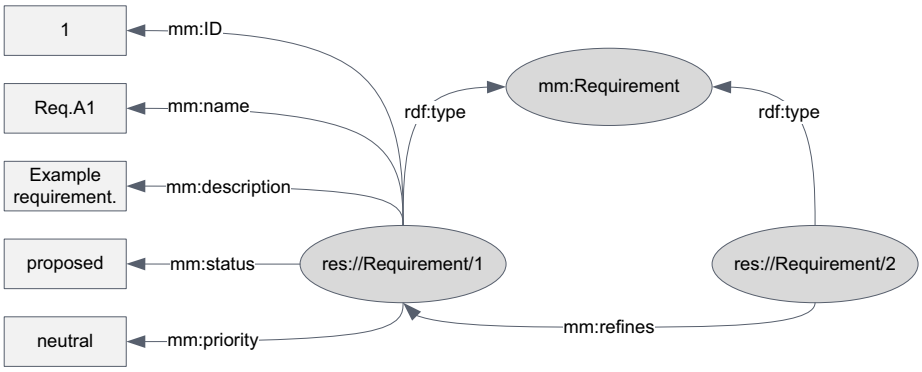

Figure 7.1: RDF notation of a requirements model

The JENA Semantic Web Framework offers an OWL API in Java to create and manipulate OWL models. By using Data Access Objects (DAO's) we separate OWL data from the application and guarantee the model remains expressed in valid OWL syntax.

7.2.3 Mapping of requirement and relation formalizations to OWL

The formalization of requirements and requirements relations are described in Chapter 4. Requirement relations are expressed in first-order logic. We use a OWL Description Logics reasoner to reason on modeled requirements using first-order logic.

Requirement relations can be inferred using the logical property characteristics of the relations (i.e. symmetry, reflexivity and transitivity) and the relation between the sets of systems satisfying the requirements. Requirement relations are also expressed in formula relations. While these formula relations can be used to infer requirement relations as well, they do no infer additional relations.

Therefore we do not apply these formula relations during the inference process. For the consistency checking process, the formula relations are essential to detect all inconsistencies. We explained this in Section 4.6.2.

Table 7.1: Logical property characteristics of relationship types

Relation type	Reflexive	Symmetric	Transitive
R1 requires R2	no	no	yes
R1 refines R2	no	no	yes
R1 contains R2	no	no	yes
R1 conflicts R2	no	yes	no
R1 partial-refines R2	no	no	yes

Table 7.1 lists the logical property characteristics of the relation types in the core metamodel, as defined by Göknil et al. [19]. It summarizes the relationship types discussed in Chapter 4.

These property characteristics are defined in OWL as specializations of OWL Object Properties. In the requirements metamodel expressed in OWL we defined the requirements relationship types as Object Properties and applied the corresponding logical property characteristics to each relationship type. The OWL reasoner uses this property characteristics to infer relations based on transitivity and symmetry without the need for additional inference rules.

In the sections describing the implementation of the Inference Engine (Section 7.4) and the Consistency Checking Engine (Section 7.5) we describe how we mapped the sets of satisfying systems and the formula relations to OWL.

7.3 The Modeling environment

The modeling environment is the bridge between the user interface and internal components. It controls the engines and provides access to the data layer through the JENA Semantic Web framework (as depicted in Figure 6.1).

7.3.1 Integration with Eclipse RCP framework

We use the Eclipse Rich Client Platform (RCP) framework as the user-interface for TRIC. The framework offers a standardized method to develop so-called Views and Editors. In TRIC there is a view listing the modeled requirements, and an editor to edit a single requirement. A construction similar to Model-View-Controller (MVC) is applied to ensure that the requirements view is synchronized with the requirements model.

The ModelingEnvironment class registers a ModelChangeListener on the JENA Ontology representing the requirements model. On change, the ModelingEnvironment in its turn notifies the registered user-interface components. As shown in Figure 7.2 the Eclipse component then retrieves a new list of all requirements in the model. This approach avoids a tight coupling between the UI layer and the data layer.

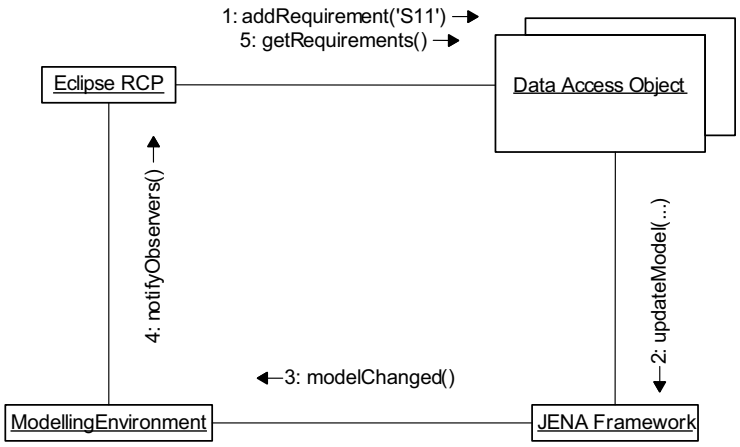

Figure 7.2: Communication diagram showing an update of the model.

7.3.2 Storage of models

When a new requirements model is created, a new OWL ontology is created using the OWL API of the JENA framework. This model is stored and manipulated in-memory. TRIC allows the user to save the requirements model to the file system. This is done by serializing the in-memory model to RDF/XML .

Internally there are three models used:

1. **The requirements model** containing the modeled requirements and relations. Conforms to the requirements metamodel.

2. **The inference model** is a copy of the requirements model. Extra elements are added which are necessary for the inference process. Afterwards contains both given and inferred requirements relations.

3. **The consistency model** is a copy of the inference model. It therefore contains the results of the inference model. Extra elements are added to support the consistency checking process.

When a model is stored to the file system only the requirements model is stored. When the Inference Engine is started the inference model is created. And at the start of the Consistency Checking Engine the consistency model is created.

7.4 The Inference Engine

The responsibility of the inference engine is to infer implicit requirements relations by applying inference rules on given requirements relations.

Section 7.4.1 describes how we represent the sets of satisfying systems to OWL description logics. In Section 7.4.2 we describe the inference process. Section 7.4.3 describes the preparation of the requirements model for inference. In Section 7.4.4 we give an overview of the reasoner rules for inferencing and explain how we derived these rules. The *partial refines* relation needs a specific approach for inferencing (see also Chapter 4 about the formalization of

requirements relations). We discuss our approach for this relation separately in Section 7.4.5. In Section 7.4.6 the analysis of derivation traces is described.

7.4.1 Representation of sets of systems

To be able to infer relations using the systems relations we need a OWL representation of the sets of systems satisfying a requirement. To realize this representation, we create a System class. Each instance of this class represents a *set of systems*. During the preprocessing of the requirements model, a system instance is created for each requirement. This system instance is related with the requirement through a **satisfies** relation.

The satisfies relation is defined as an OWL Object Property. The RDF description is provided in Listing 7.1. The source of a satisfies relation is a system (rdfs:domain), the target is a requirement (rdfs:range). Because the domain and range of the relation differ, the satisfies relation is non-reflexive, non-symmetric and non-transitive.

```
<rdf:Description rdf:about="satisfies">
 <rdfs:range rdf:resource="Requirement"/>
 <rdfs:domain rdf:resource="System"/>
 <rdf:type rdf:resource=
        "http://www.w3.org/2002/07/owl#ObjectProperty"/>
</rdf:Description>
```

Listing 7.1: RDF definition of the *satisfies* relation

In the section about preparing the requirements model for inference (Section 7.4.3) we continue on this relation.

Reasoning on sets of systems

The formalization approach can infer the following (implicit) relations, based on sets of systems related to requirements. We assume that the set of systems S1 satisfies requirement R1 and the set of systems S2 satisfies requirement R2.

- **R1 requires R2** iff S1 $\subset$ S2
- **R1 conflicts R2** iff S1 $\cap$ S2 $= \varnothing$ (S1 is disjoint with S2)

In OWL-DL, classes cannot be related to individuals (instances), and a *subClassOf* or *disjointWith* statement is only valid between two classes [11]. To relate requirements (modeled as individuals) with sets of systems, we need individuals of systems. Since we need to state a subset relation and express disjointness of sets of systems, we defined these relations as Object Properties to overcome this limitation of OWL-DL . To enable correct reasoning about subsets and disjoint sets we need to specify their details.

The `subSetOf` relation is listed in Listing 7.2. It is a *transitive* relation by definition.

```
<rdf:Description rdf:about="subSetOf">
  <rdfs:range rdf:resource="System"/>
  <rdfs:domain rdf:resource="System"/>
  <rdf:type rdf:resource=
    "http://www.w3.org/2002/07/owl#TransitiveProperty"/>
```

```
</rdf:Description>
```

Listing 7.2: RDF definition of the *subSetOf* relation

The `disjointWith` relation is listed in Listing 7.3. It is a non-reflexive, symmetric and non-transitive relation between two systems. The symmetric property is not represented in OWL, because we noticed that the reasoner did not reason properly over symmetric object properties. To overcome this, we apply a `disjointWith` relation in both directions when a *conflicts* relation is encountered.

```
<rdf:Description rdf:about="disjointWith">
  <rdfs:range rdf:resource="System"/>
  <rdfs:domain rdf:resource="System"/>
  <rdf:type rdf:resource=
     "http://www.w3.org/2002/07/owl#ObjectProperty"/>
</rdf:Description>
```

Listing 7.3: RDF definition of the *disjointWith* relation

7.4.2 The inference process

The inference process takes the requirements model as input and infers implicit requirements relations based on given relations and a set of inference rules. As explained in Section 6.3, we chose to separate the inference process from the consistency checking process. Hence we infer all implicit relations, even if the given requirements model is inconsistent already.

We identify the following steps in the inference process:

1. **Preparation of the inference model:** create a new model from the given requirements model, and add data to the model necessary for inferencing.
2. **Retrieval of inference rules:** locate and import the inference rules belonging to the used metamodel from the file system.
   These rules are divided into three categories:
   (a) a rule expressing the property of the subset relation
   (b) rules mapping requirements relations to relations between sets of systems
   (c) rules specific for the *partial-refines* relation
3. **Execution of reasoner:** with the preprocessed requirements model and the inference rules as input, implicit relations are inferred and added to the model. These inferred relations become visible in the user-inter
4. **Derivation trace analysis:** optional step in the process, provides a derivation trace of an inferred relation.

In the next sections we will describe these steps and elaborate on the mapping between the formalization of requirements and relations and the inference rules for the reasoner.

7.4.3 Preparation of the inference model

The requirements model created by the requirements engineer only contains requirements and relations between them. As mentioned before, to infer relations we need a representation of the sets of systems that satisfy the requirements. The ontology containing the requirements model is taken as input, and copied to a new ontology used for inference (the inference model). Then, for each requirement a system instance is added.

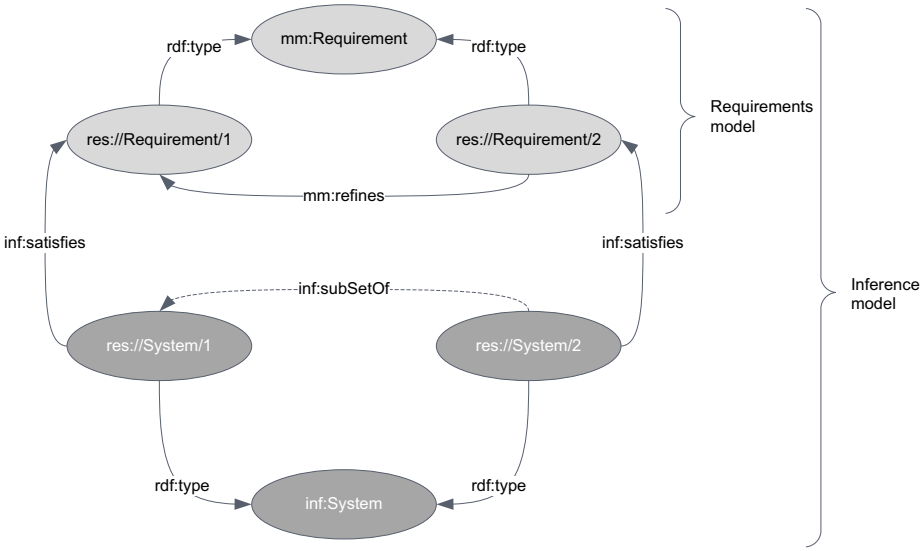

Figure 7.3: RDF representation of the inference model

Figure 7.3 shows the relation between the requirements model and the inference model. In addition to the requirements model, the inference model contains System instances. These systems are related with requirements through a *satisfies* object property. The dashed `subSetOf` relation between both Systems is to illustrate the result *after* execution of the reasoner. The subset relations are not applied during the preparation.

7.4.4 Reasoner rules for inferencing

In the formalization (Chapter 4) requirements relations are formalized using first-order logic (FOL) and basic Set Theory. Because the logical property characteristics of the relations are defined in the (imported) metamodel ontology, the JENA reasoner already infers implicit relations based on transitivity using the builtin OWL reasoner rules.

In Section 7.2.3 we discussed the mapping of requirement relations to relations between sets of systems. Table 7.2 gives an overview of the rules that establish this mapping. S1 and S2 are the sets of systems satisfying requirements R1 and R2 respectively. Note that these rules are specific for the core requirements metamodel, when a customized metamodel is used, the metamodel engineer needs to customize these rules as well.

Table 7.2: Overview of inference rules for the core metamodel

Rule name	Matching term	Conclusion
`requires_to_subset`	R1 *requires* R2	S1 is a subset of S2
`refines_to_subset`	R1 *refines* R2	S1 is a subset of S2
`contains_to_subset`	R1 *contains* R2	S1 is a subset of S2
`subset_to_requires`	S1 is a subset of S2	R1 *requires* R2
`conflicts_to_disjoint`	R1 *conflicts* R2	S1 and S2 are disjoint[1]
`disjoint_to_conflicts`	S1 and S2 are disjoint[1]	R1 *conflicts* R2
`temp_req_to_p_ref`	See Section 7.4.5.	

[1] i.e. $(S1 \cap S2) = \varnothing$

As mentioned before, the Inference Engine will infer any relation, even if the result is an inconsistency. If for example R1 *requires* R2, and R1 *conflicts* R2, the reasoner rules will conclude that S1 $\subset$ S2 *and* $S1 \cap S2 = \varnothing$. This of course is a contradiction, but we use the Consistency Checking Engine to detect this.

Syntax of JENA reasoner rules

In this section all the inference rules are listed. The rules are expressed in JENA reasoner rule syntax. The simplified syntax is given in Listing 7.4.

```
Rule       :=  [ ruleName : bare-rule ]

bare-rule  :=   term, ... term -> term, ... term

term       :=   (node, node, node)
           or   (node, node, functor)
           or   builtin(node, ... node)

functor    :=   functorName(node, ... node)

node       :=   uri-ref
           or   prefix:localname
           or   <uri-ref>
           or   ?varname
           or   'a literal'
           or   'lex'^^typeURI
           or   number
```

Listing 7.4: JENA reasoner rule syntax (simplified)

If terms are matched by the first part of the rule, the terms following `'->'` are concluded (inferred). Variables are denoted with a '?'. Variables are not typed. A variable will match with any node in the model, which could be requirements or systems (resources) or relations (object properties). In our model requirements are related through object properties. To ensure a variable has a certain type we could have added the following line:

```
(?r1 rdf:type Requirement)
```

However in our case we know that when the following term matches:

```
(?r1 refines ?r2)
```

the variables `r1` and `?r2` *must* be instances of Requirement. In our tool only requirements can be related through a *requires* relation. Analogously we assume that if a `satisfies` is matched in a triple, the left-hand side `?s1` is a system instance, and the right-hand side `?r1` is a requirement instance:

```
(?s1 satisfies ?r1)
```

Therefore, we do not need to check explicitly for the variable's type in the reasoner rules.

Inference rules for the core requirements metamodel

We provide the reasoner rules for the inference engine and discuss the rules grouped by functionality. The complete list of inference rules in JENA reasoner rule syntax can be found at Appendix B.

```
@include <OWL>.

[subset_also_disjoint:   (?s1 inf:subSetOf ?s2)
                         (?s2 inf:disjointWith ?s3)
                     ->
                         (?s1 inf:disjointWith ?s3)]
```

Listing 7.5: Import of standard OWL rules and rule needed for disjoint subsets

In Listing 7.5 the initial rules are listed. The first line starts with `@include <OWL>.`, which tells the reasoner to import the rules for OWL-DL. This enables the reasoner to reason on OWL and RDF constructs such as transitive object properties.

The Object Properties in the rule's terms are prefixed with 'inf:'. This prefix refers to the inference model. Because the *subSetOf* and *disjointWith* relation are added to the inference model.

Because we defined our own 'subSetOf' and 'disjointWith' relation between sets of systems, we need an additional rule to ensure the so-called *'permeation of disjointness'*, which states that subsets of disjoint sets are also disjoint:

$$(S_1 \subset S_2) \wedge (S_2 \cap S_3 = \varnothing) \Rightarrow (S_1 \cap S_3 = \varnothing)$$

This is expressed in the rule `subset_also_disjoint`.

The next set of rules in Listing 7.6 map the *requires, refines* and *contains* relation to a subset relation between their corresponding systems.

Since we query for the linked requirements using the `satisfies` property, `?s1` is the set of systems satisfying requirement `?r1` and `?s2` the set of systems satisfying requirement `?r2`.

```
[requires_to_subset:     (?r1 mm:requires ?r2)
                         (?s1 inf:satisfies ?r1)
                         (?s2 inf:satisfies ?r2)
                     ->
                         (?s1 inf:subSetOf ?s2)]

[refines_to_subset:      (?r1 mm:refines ?r2)
                         (?s1 inf:satisfies ?r1)
```

```
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:subSetOf ?s2)]

[contains_to_subset:    (?r1 mm:contains ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:subSetOf ?s2)]
```

Listing 7.6: Rules mapping requirement relations to subset relations between systems

When the set of systems satisfying a requirement is a subset of the set of systems satisfying another requirement we infer a *requires* relation. This rule is listed in Listing 7.7.

```
[subset_to_requires:    (?s1 inf:subSetOf ?s2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?r1 mm:requires ?r2)]
```

Listing 7.7: Rule mapping a subset (of systems) to a requires relation

If there exists a conflicts relation between requirements R1 and R2, then their sets of systems are disjoint (i.e. there is no system satisfying both R1 and R2). The other way around also holds: if sets of systems are disjoint, the requirements they satisfy are conflicting. These rules are mapped to two reasoner rules as listed in Listing 7.8. The concluding terms of the first rule (`conflicts_to_disjoint`) are stating a `disjointWith` relation in both directions because the symmetry of the disjointWith property is not handled properly by the JENA reasoner (as mentioned before in Section 7.4.1).

```
[conflicts_to_disjoint: (?r1 mm:conflicts ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:disjointWith ?s2)
                        (?s2 inf:disjointWith ?s1)]

[disjoint_to_conflicts: (?s1 inf:disjointWith ?s2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?r1 mm:conflicts ?r2)]
```

Listing 7.8: Rules to map conflicts to disjointness of systems and vice versa. between systems

7.4.5 Dealing with the partial refines relation

The *partial refines* relation needs a specific approach since it is a special combination of the *refines* and *contains* relations. In the formalization, the partial refines relation is decomposed into *refines* and *contains* relations with a special temporal requirement.

There are two decompositions which both are applied:

(a) contains(R_2, R_{T12}) and refines(R_1, R_{T12}) iff partial-refines(R_1, R_2)

(b) refines(R_{T12}, R_2) and contains(R_{T12}, R_1) iff partial-refines(R_1, R_2)

These temporal requirements are created during the **preprocessing** step. Figure 7.4 depicts the decomposition of the *partial refines* relation.

Because we need to distinguish the temporal requirement from the given requirements, we added a *data type property* to the Requirement type. When `isTemporal` is set to true, it is a temporary requirement. The rules in Listing 7.9 only match on a temporary requirement. The `isTemporal` literal is also used to filter out the temporary requirement in the user interface. The requirements engineer should not be able to modify it.

The rules in Listing 7.9 are of a different type than the other rules. These rules use a `'<-'`, and have the concluding terms before the matching terms. This type of rule is called a *backward rule*, as opposed to the forward rules. Backward rules can be seen as 'goal-driven' rules because they match and execute when the reasoning engine queries to satisfy a certain goal. Forward rules are 'data-driven'. They trigger on given data to infer new triples.

Most reasoners use either forward or backward rules. The JENA reasoner offers a *hybrid mode* combining forward and backward rules. In the hybrid mode forward rules are executed before execution of the backward rules. Therefore the backward rules operate on the model including inferred statements by the forward rules. The rule for partial refine should work using a forward rule, but we noticed it did not resulted in a correct inference.

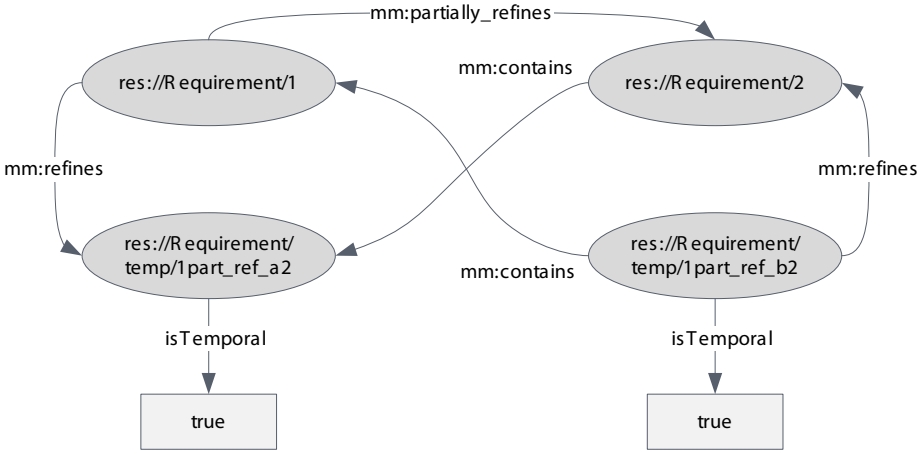

Figure 7.4: Decomposition of the partial refines relation using temporary requirements

```
[temp_req_to_p_ref1: (?r1 partial_refines ?r2)
                     <-
                      (?r1 refines ?rt)
                      (?r2 contains ?rt)
                      (?rt isTemporary 'true'^^xsd:boolean)]
```

```
[temp_req_to_p_ref2: (?r1 partial_refines ?r2)
                     <-
                      (?rt contains ?r1)
                      (?rt refines ?r2)
                      (?rt isTemporary 'true'^^xsd:boolean)]
```

Listing 7.9: Inference rules for partial refines

Using the term `(?rt isTemporary 'true'^^xsd:boolean)` we make sure the variable `?rt` is bound to a temporary requirement.

7.4.6 Derivation trace analysis

The main reason of the choice of the JENA Semantic Web Framework is that the reasoner keeps a log of its derivations. Internally JENA uses a graph-structure to perform the reasoning. When logging is enabled, it keeps this structure to be inspected afterwards. Furthermore, it creates RuleDerivation objects containing specific information about which rule is fired and which triples are matched.

The provided RuleDerivation class, however, is very basic. It allows textual output of a derivation trace. To visualize the derivation traces, we needed a more specific approach. Therefore, we created a subclass of RuleDerivation. This subclass (ExtendedRuleDerivation) allows access to the data in the inferred model. We adopt the code for the textual output to collect the involved requirements and relations in a derivation trace. We use a Java object as a data type to represent the triples encountered:

$$Relationship ::= \langle requirement_{source}, relation, requirement_{target} \rangle$$

We define a **derivation trace** as an unordered set of relationship triples involved in the derivation of a certain relationship. During inference a relationship can be used multiple times to derive an inferred relation. Therefore, it occurs multiple times in a single derivation trace. We use a Java Set implementation to avoid these duplicate entries in a trace since a Set cannot contain duplicate entries by definition. As a result, a set of derivation traces is created. However, some traces are equal with respect to the involved relationship triples (as the order of the triples is not relevant).

To filter out these duplicate traces, we again make use of a Set implementation. A set of derivation traces (being sets of relationship triples) is returned.

$$Return\ type ::= \{\{Relationship\}\}$$

These traces are available for the Visualization Engine, which is discussed in Section 7.6.

Figure 7.5 gives an example of multiple traces for a single inferred relation. The model consists out of five requirements. There are three unique traces which derive 'R1 requires R4', although trace 1 and 3 are using the same path. The textual output is listed in Listing 7.10.

```
[DERIVATION LOG]--------------------------
Statement is (res://Requirement/1 requires res://Requirement/4)
Number of traces: 8, number of unique traces: 3
```

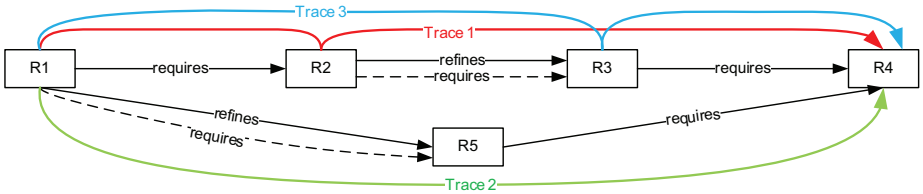

Figure 7.5: Multiple traces which derive 'R1 requires R4'

```
Trace number: 1
    -R1 requires R2 (given)
    -R2 refines R3 (given)
    -R2 requires R3 (inferred)
    -R3 requires R4 (given)
    -R2 requires R4 (inferred)
    -R1 requires R4 (inferred)

Trace number: 2
    -R1 refines R5 (given)
    -R1 requires R5 (inferred)
    -R5 requires R4 (given)
    -R1 requires R4 (inferred)

Trace number: 3
    -R1 requires R2 (given)
    -R2 refines R3 (given)
    -R2 requires R3 (inferred)
    -R1 requires R3 (inferred)
    -R3 requires R4 (given)
    -R1 requires R4 (inferred)
```

Listing 7.10: Output of derivation log

The difference between trace 1 and 3 is the path followed over inferred relationships. In trace 1 'R2 requires R4' is used, while in trace 3 'R1 requires R3' is used.

7.5 The Consistency Checking Engine

The design of the Consistency Checking Engine is very similar to the Inference Engine because first-order logic is also used to detect inconsistencies. The term "consistency" can be confusing because it is used in OWL as well. Therefore, it is important to stress that we check for consistency of the modeled requirements using metamodel specific consistency rules. This is different from checking whether the model itself is expressed in valid OWL syntax. The latter is ensured by using TRIC for modeling the requirements.

7.5.1 The consistency checking process

The consistency checking engine needs the inferred requirements model as input because some consistency rules need the 'sets of systems' formalization.

As explained in Section 6.3, checking for consistency without inferring implicit relations can result in a wrong conclusion.

The consistency checking process has the following steps:

1. **Preprocessing of the inferred model:** take the inferred model as input, and add the formula relations (object properties) to the model.

2. **Retrieval of consistency rules:** locate and import the consistency rules belonging to the used metamodel from the file system.
   These rules are divided into three categories:

   (a) rules mapping requirements relations to formula relations

   (b) rules expressing the properties of the formula relations

   (c) consistency checking rules, which match with inconsistencies

3. **Registering of inconsistency functor:** to report inconsistencies, a custom functor is registered.

4. **Execution of reasoner:** with the preprocessed requirements model and the consistency rules as input, the inconsistency functor is called for each inconsistency encountered.

We explain these steps in the next sections, and provide more detail about the reasoner rules for consistency checking.

7.5.2 Preprocessing of the inferred model

We take the inferred model as input for the Consistency Checking Engine. The preprocessing is analogous to the preprocessing of the Inference Engine. Object properties representing formula relations are added to the model.

In the inference engine we added system instances to represent the sets of systems satisfying a requirement. For the Consistency Checking Engine we need the formula relations between the property representation of requirements to check for consistency. Unlike the Inference Engine we do not create separate property objects to represent these properties. We directly relate requirement instances with formula relations.

There are five formula relations in the formalization for the core metamodel:

- all-in-whole
- all-in-part
- all-implies-in
- some-implies-in
- all-equals-in

These relations are discussed in Chapter 4.

In Listing 7.11 an example is given of a formula relation expressed in OWL. The formula relations are defined as object properties with the Requirement class as range and domain.

```
<owl:TransitiveProperty rdf:ID="all_in_whole">
  <rdfs:range rdf:resource="mm:Requirement"/>
  <rdfs:domain rdf:resource="mm:Requirement"/>
  <rdf:type rdf:resource=
    "http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:TransitiveProperty>
```

Listing 7.11: The all-in-whole formula relation as object property

The mapping of requirement relations to formula relations is done with reasoner rules together with the execution of the other rules for the consistency checker.

7.5.3 Reasoner rules for consistency checking

In this section we elaborate on the reasoner rules for the consistency checking engine. Each set of rules is discussed separately. A complete list of the rules for the consistency checker can be found in Appendix C.

The rules for consistency checking use both the sets of systems satisfying requirements and the property representation of requirements. The satisfying sets of systems are defined and inferred in the Inference Engine. In the Consistency Checking Engine we need to add the formula relations between requirements.

The first step is to map the requirements relations to formula relations. In Section 4.5.2 we provided this mapping. As mentioned in the previous section, we use the formula relations directly between requirements.

While the mapping is in both directions (from requirements relations to formula relations and vice versa), we do not apply the mapping from formula relations to requirements relations. The requirements relations are already inferred by the Inference Engine.

In Listing 7.12 the reasoner rules creating this mapping are listed. The prefix `cons:` refers to the namespace of the consistency checker where the object properties for the formulas are declared in that namespace.

```
[map_refines_to_formulas:  (?r1 mm:refines ?r2)
                        ->
                           (?r1 cons:all_in_whole ?r2)
                           (?r1 cons:some_implies_in ?r2)]

[map_contains_to_formulas: (?r1 mm:contains ?r2)
                        ->
                           (?r2 cons:all_in_part ?r1)
                           (?r2 cons:all_equals_in ?r1)]

[map_part_ref_to_formulas: (?r1 mm:partial_refines ?r2)
                        ->
                           (?r1 cons:all_in_part ?r2)
                           (?r1 cons:all_implies_in ?r2)]
```

Listing 7.12: Reasoner rules mapping requirements relations to formula relations

Second, the properties of the formula relations (see Chapter 4) are applied. These rules are listed in Listing 7.13.

```
[formula_rule_1:    (?p1 cons:all_in_part ?p2)
                    (?p2 cons:all_in_whole ?p3)
```

```
                ->
                    (?p1 cons:all_in_part ?p3)]

[formula_rule_2:    (?p1 cons:all_in_whole ?p2)
                    (?p2 cons:all_in_part ?p3)
                ->
                    (?p1 cons:all_in_part ?p3)]

[formula_rule_3:    (?p1 cons:some_implies_in ?p2)
                    (?p2 cons:all_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_4:    (?p1 cons:all_implies_in ?p2)
                    (?p2 cons:some_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_5:    (?p1 cons:some_implies_in ?p2)
                    (?p2 cons:all_equals_in ?p3)
                ->
                    (?p1 cons:some_implies_in ?p3)]

[formula_rule_6:    (?p1 cons:all_implies_in ?p2)
                    (?p2 cons:all_equals_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_7:    (?p1 cons:all_equals_in ?p2)
                    (?p2 cons:all_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]
```

Listing 7.13: Reasoner rules expressing the formula relation properties

From the formalization we derived a set of contradicting facts resulting in seven inconsistency rules.

- **Contradicting subset relations**
  In the formalization a *proper* subset relation is used between sets of systems satisfying requirements. This implies that a set of system cannot be a subset of itself (inconsistency 1). Furthermore two sets of systems S1 and S2 cannot have a subset relation with each other: $(S1 \subset S2) \wedge (S2 \subset S1)$ contradicts (inconsistency 2). Because of transitivity of the subset relation the first case $(S1 \subset S1)$ will be inferred when two systems are a subset of each other. We put the second case as an additional rule because it provides more information about which requirements caused the contradicting subset relations.

- **Contradicting requirement relations**
  By definition a *requires* relation contradicts a *conflicts* relation (inconsistency 3). Note that the Inference Engine infers a *requires* relation given a *refines* or *contains* relation. Therefore we do not create separate rules for each of them.

When using the subset relations and disjointness between sets of system we can detect the same situation (i.e. $S1 \subset S2$ contradicts $S1 \cap S2 = \varnothing$). We use the *requires* and *conflicts* relations to detect the inconsistency in our rule.
The *partial-refine* relation also contradicts with a *conflicts* relation (inconsistency 4).

- **Contradicting formula relations**
  The formalization of formula relations between the properties of requirements state disjointness between a number of them:
  - *all-in-part* and *all-in-whole* are disjoint. Hence ($R1$ *all-in-part* $R2$) $\wedge$ ($R1$ *all-in-whole* $R2$) is contradicting (inconsistency 5).
  - *all-equals-in* and *all-implies-in* are disjoint. Hence ($R1$ *all-equals-in* $R2$) $\wedge$ ($R1$ *some-implies-in* $R2$) is contradicting (inconsistency 6).
  - *all-equals-in* and *some-implies-in* are disjoint. Hence ($R1$ *all-equals-in* $R2$) $\wedge$ ($R1$ *some-implies-in* $R2$) is contradicting (inconsistency 7).

We defined reasoner rules to match these seven inconsistencies. These rules are listed in Listing 7.14. Each rule calls the `addInconsistency`-functor to report the inconsistency properly to the user. This custom functor is discussed in the next section.

```
[inconsistency_1: (?s1 inf:subSetOf ?s1)
                  (?s1 inf:satisfies ?r1)
                ->
   addInconsistency('Circular dependency',?r1)]

[inconsistency_2: (?s1 inf:subSetOf ?s2)
                  (?s2 inf:subSetOf ?s1)
                  notEqual(?s1,?s2)
                  (?s1 inf:satisfies ?r1)
                  (?s2 inf:satisfies ?r2)
                ->
   addInconsistency('Contradicting subsets of systems',
                        ?r1,?r2)]

[inconsistency_3:  (?r1 mm:conflicts ?r2)
                   (?r1 mm:requires ?r2)
                ->
   addInconsistency('Both conflicts and depends (req.)',
                        ?r1,?r2)]

[inconsistency_4: (?r1 mm:conflicts ?r2)
                  (?r1 mm:partial_refines ?r2)
                ->
   addInconsistency('Both conflicts and depends (part.ref.)',
                        ?r1,?r2)]

[inconsistency_5: (?r1 cons:all_in_part ?r2)
                  (?r1 cons:all_in_whole ?r2)
                ->
   addInconsistency('Requirement both part-of and whole',
```

```
                        ?r1,?r2)]

[inconsistency_6: (?r1 cons:all_equals_in ?r2)
                  (?r1 cons:all_implies_in ?r2)
                ->
   addInconsistency('all_equals_in contradicts all_implies_in',
                        ?r1,?r2)]

[inconsistency_7: (?r1 cons:all_equals_in ?r2)
                  (?r1 cons:some_implies_in ?r2)
                ->
   addInconsistency('all_equals_in contradicts some_implies_in',
                        ?r1,?r2)]
```

Listing 7.14: Inconsistency rules for the core metamodel

7.5.4 Detecting inconsistencies

A custom inconsistency functor is added to the reasoner engine. A functor is a function which can be invoked inside a reasoner rule. When an inconsistency is matched by a rule, this functor is called. The parameters of the functor can provide extra information, such as a textual description and references to the involved requirements. Each rule listed in Listing 7.14, uses this functor.

When the `addInconsistency`-functor is called, the inconsistency will be shown in the Console as textual output. Furthermore, the Consistency Checker is notified of the event that an inconsistency is found. When the reasoner is done, a dialog box prompts the user about the consistency of the requirements model. This is either *"The model is consistent."* or *"One or more inconsistencies are found."*.

The `addInconsistency`-functor takes a flexible amount of arguments. The first argument is a string containing the message to display to the user, followed by one or more references to requirement resources. This is useful because some inconsistencies may occur on a single requirement (for instance `R1 requires R1`), but it may also be a collection of requirements (and relations) being inconsistent.

```
[Inconsistency]
Reason: 'Contradicting subsets of systems'
Involved requirements:
    - res://Requirement/3
    - res://Requirement/1
```

Listing 7.15: Example output of an inconsistency

Listing 7.15 gives an example of the output for a single inconsistency. In future work, the implementation of this functor can be extended to support analysis of inconsistencies.

7.6 The Visualization Engine

The responsibility of the Visualization Engine is to provide a visual explanation of the reasoner's results. There are three steps we take in our implementation

to achieve this: first, we use the derivation logging functionality of JENA to collect all traces leading to the facts (given relations) causing a specific relation to be inferred. Second, we reduce the amount of traces to unique paths. And finally we visualize one of the unique traces by using JGraph.

The first two steps are performed by the Inference Engine, the process of collecting the derivation traces is described in Section 7.4.6.

JGraph [3] is a powerful and easy-to-use open source package. We use it to visualize derivation traces. A derivation trace is a set of Relationship objects, which represent a 3-tuple:

$$\langle requirement_{source}, relation, requirement_{target} \rangle$$

In Section 7.4.6 we described how we create a collection of unique derivation traces. In the current implementation, we visualize the first derivation trace.

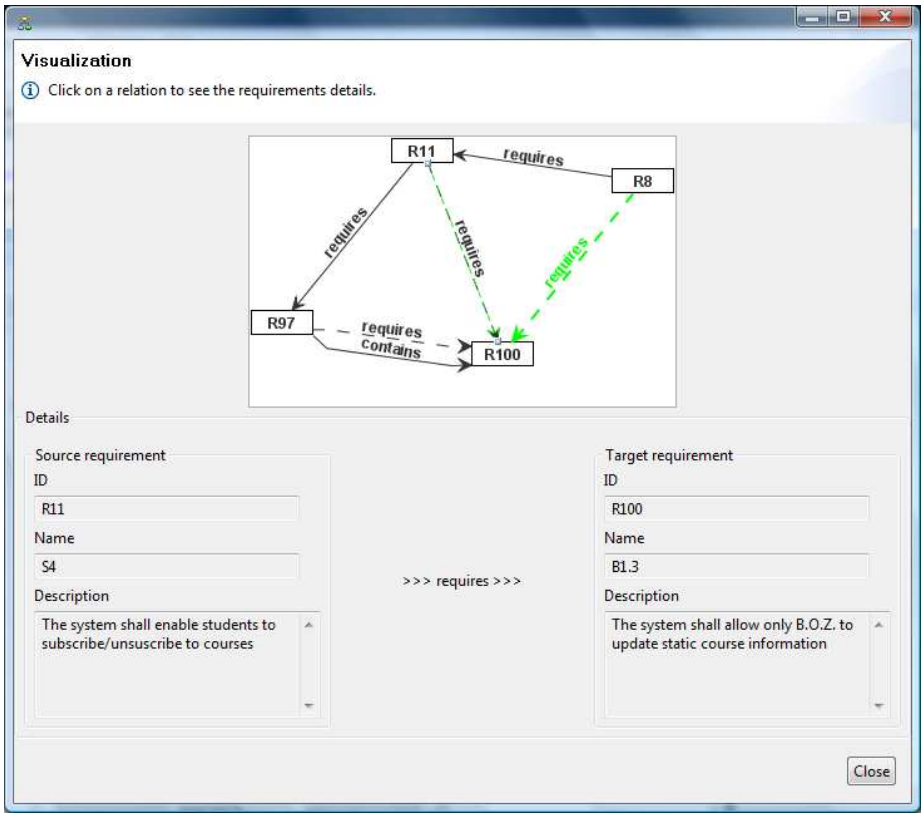

Figure 7.6: Screenshot of the Visualization Engine output

For each requirement a node is created, and for each relation an edge. The output is shown in Figure 7.6. Different graphical styles are applied to distinguish inferred relations from given relations. Inferred relations are shown with a dashed line. The relation to explain is dashed and colored green.

A selection listener is registered on the graph. When the user clicks on an edge (relation), the related requirements are retrieved and shown. This enables the requirements engineer to quickly inspect the requirements and to analyze the

inferred relations. Because both details of the source and target requirement are provided, the correctness of the given and inferred relations can be evaluated.

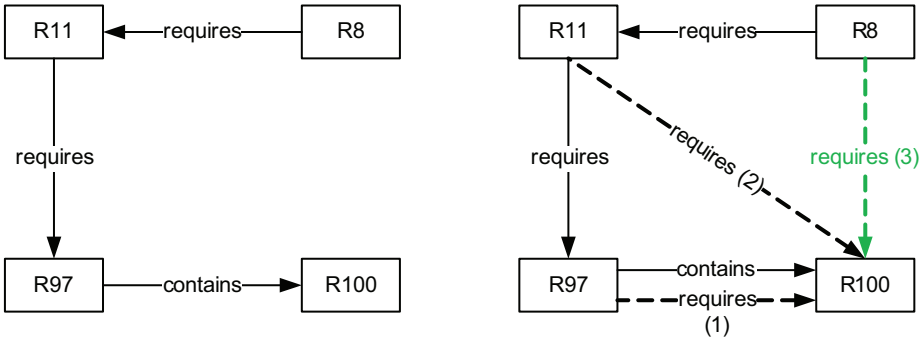

Figure 7.7: Derivation steps to derive 'R8 requires R100'

Figure 7.6 shows a screenshot of the Visualization dialog, and illustrates how the *requires* relation is derived between R8 and R100.

In Figure 7.7 the sequence of steps taken by the reasoner is shown:

1. The *requires* relation between R97 and R100 is derived from the *contains* relation

2. Because of transitivity of the *requires* relation a *requires* relation between R11 and R100 is inferred (R11 *requires* R97, R97 *requires* R100, hence R11 *requires* R100).

3. Again by transitivity R8 *requires* R100 (through R11).

7.7 Usage of the tool

This section describes the usage of TRIC. Some screenshots are provided to illustrate the graphical user-interface, which is implemented using Eclipse RCP.

Figure 7.8 shows the main window of TRIC. On the left-hand side there is a view listing all the requirements in the model. On the right-hand side there is an editor enabling the user to modify an individual requirement. When modifying a requirement in the editor, the list of requirements is automatically updated. By using the `File` menu, users can open, save and create new models.

7.7.1 Adding requirements to the model

Requirements are added to the model via a dialog. The dialog (shown in Figure 7.9) allows the requirements engineer to add a new requirement. It ensures that all required attributes are entered. When for example the name of a requirement is omitted, the dialog will display an error message. This prevents the creation of a requirement which does not conform to the metamodel.

Each requirement is assigned a unique ID which cannot be chosen or modified by the user. The ID is used internally to distinguish and sort requirements. The requirement's name attribute can be used by the requirements engineer as

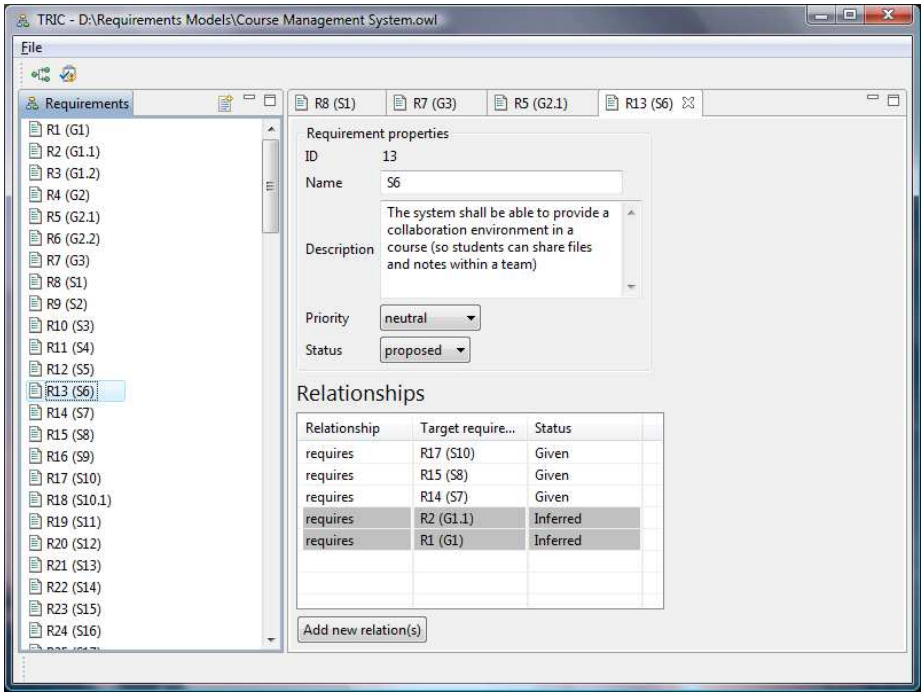

Figure 7.8: Screenshot of TRIC's main window

a more flexible identifier (as these will vary between requirements specification documents).

Deleting requirements is not possible in the prototype.

7.7.2 Relating requirements

Once entered, requirements can be related to other requirements. In the requirements editor related requirements are listed (visible in Figure 7.8). A dialog is provided to relate a requirement to other requirements. This dialog is depicted in Figure 7.10. For each relationship type in the metamodel, a button is created. This enables the requirements engineer to use all available relationship types. Relating a requirement to itself is prevented by the interface since this does not make sense or introduces an inconsistency (for example a requirement which refines itself).

If relations are assigned erroneously (which the requirements engineer may discover after analysis of the inferred relations), they can be deleted. A pop-up menu is available when right-clicking on one of the listed related requirements.

7.7.3 Inference results

After the inference process all modeled requirements are listed textually in the console. For each requirement the related requirements are provided, both given and inferred relations. An example of this textual output is shown in Listing 7.16.

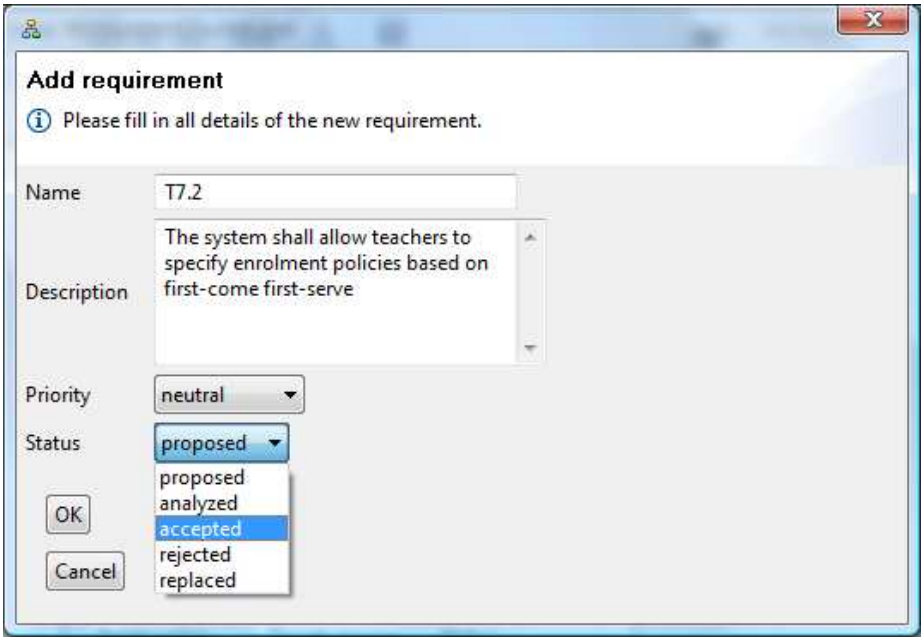

Figure 7.9: Dialog to add a new requirement

```
The following requirements are modelled:

-Requirement 1(G1):
-Requirement 2(G1.1):
    >> refines >> 1(G1)
    >> requires >> 1(G1)
-Requirement 3(G1.2):
    >> refines >> 1(G1)
    >> requires >> 1(G1)
    >> requires >> 2(G1.1)
```

Listing 7.16: Textual output given at end of the inference process

After the Inference Engine is done it will make a call to the `notifyObservers()`-method of ModelingEnvironment. This causes the user-interface components to refresh their content. The inferred relations appear in the requirements editor. Given and inferred relations are distinguished by using different background colors. Furthermore a column in the table denotes whether a relation is either 'given' or 'inferred'. See Figure 7.8 on page 57 for an example.

The requirements engineer is now able to explore the inferred relations by browsing through the modeled requirements.

When right-clicking on a row a popup menu shows up, which offers the following actions:

Show visualization opens the Visualization dialog, with a visual explanation of the inferred relation.
Intended for inferred relations only.

Show derivation log writes a textual log of the derivation traces to the con-

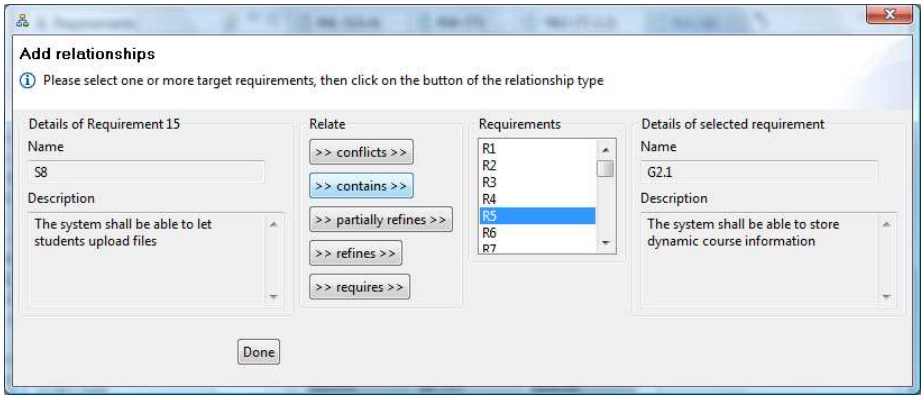

Figure 7.10: Dialog to relate requirements

sole.
Intended for inferred relations only.

Delete relationship enables the user to delete a relationship from the model, a confirmation dialog is provided.
Intended for given relations only.

7.8 Conclusion

In this chapter we described the implementation of TRIC. We use OWL as a platform for reasoning because there is a lack of reasoner engines capable of reasoning about EMF models. Because the formalization uses first-order logic OWL description logics can be used.

We created a mapping between the formalization of requirements and requirements relations (discussed in Chapter 4) and OWL. We expressed requirement relations as OWL Object Properties. This enabled us to represent the logical property characteristics (i.e. transitivity and symmetry) in the metamodel. An OWL reasoner then can infer implicit relations based on these properties without additional reasoner rules.

Because the formalization relies on sets of satisfying systems and relations between the Property representation of requirements (formula relations), we needed to represent these in OWL. The sets of satisfying systems are added to the model during a preprocessing step. The relations between the sets of systems and the formula relations are realized by reasoner rules.

By using the derivation logs of the JENA OWL reasoner we are able to visualize the path followed by the reasoner to derive a certain inferred relation.

In the next chapter we evaluate the quality of the implementation.

8 Evaluation of TRIC

8.1 Introduction

In this chapter we evaluate the tool support we developed. To evaluate the support for the modeling process, we created an example requirements document for a Course Management System (CMS) . We covered all the relationship types provided by the requirements metamodel. And we deliberately added contradicting requirements to verify the consistency checking functionality of TRIC.

Besides evaluating the functionality of the tool, we also evaluate the strengths and weaknesses of the tool design using the nonfunctional requirements. We described the requirements for the tool in Chapter 5.

First we introduce the example case of the Course Management System in Section 8.2. In Section 8.3 we evaluate the functionality of the tool using the example case. In Section 8.4 we evaluate the quality of the design and implementation. We conclude this chapter in Section 8.5.

8.2 Example case: Course Management System

To illustrate the purpose of the tool, we created an example case. A popular example in academic research is a Course Management System, since readers are familiar with the domain. In a couple of brainstorm sessions we gathered requirements for such a system. We deliberately added contradicting requirements among stakeholders. This section briefly describes the stakeholders and the content of the requirements. For the complete list of requirements and a glossary we refer to Appendix A.

8.2.1 Stakeholders

The system has four groups of stakeholders, each having a different role and usage of the system, and hence having different requirements.

Lecturer Uses the CMS to provide course material to students, and to manage the course's administration (such as grading and keeping track of deliverables).

Student Uses the CMS when participating courses. The CMS provides information about courses, and is used to subscribe to courses, hand in work, collaborate with other students, etc.

Administration or *Bureau Onderwijs Zaken* (Office for Educational Affairs) uses the CMS to manage courses, assigning teachers to courses, and coordinating the enrollment policies. Furthermore the administration collects all kinds of statistics using the CMS, such as grade averages per course.

Maintainer Provides technical support for CMS, creating and restoring backups is one of the responsibilities.

The requirements in Appendix A are grouped by the proposing stakeholder.

8.2.2 Contradicting requirements

In many cases stakeholders have different interests in a system. In our example this is the case. Especially there is a field of tension between lecturers and the administration. Lecturers desire to have full control about their courses, while the administration needs to check all the rules and do not want lecturers to bypass them. An example of two contradicting requirements is:

Lecturer:

> **R48** The system shall allow lecturers to create courses

and

Administration:

> **R98** The system shall allow *only* the administration to create new courses

8.2.3 Identification of requirements relations

After gathering requirements for the Course Management System for each stakeholder, we tried to identify all relations between requirements. We did this by thinking in terms of system components and system data. For example, in order to send messages to students, the requirement for a messaging system (**R7**) should also be fulfilled. This indicates a *requires* relation between requirements using the messaging system and the requirement for a messaging system. With system data we mean the following: when a requirements states the usage of data in the system, the system should allow the entering of that data. For example students subscribing to a course, the course should be created first. Hence there is a *requires* relation between those requirements.

Some relations were trivial to identify, because we created the requirements in order to use all relation types. For example **R61** *contains* **R62**, **R63** and **R64**.

8.3 Evaluation of tool functionality

We use the example of the Course Management System to evaluate the functionality of the tool. Because it is not an industrial requirements document we cannot use it to properly evaluate the requirements modeling and reasoning approach.

In the CMS requirements document there are 122 requirements (94 functional and 28 non-functional requirements). We started with identifying the requirements relations in the CMS case. Then we modeled the requirements and relations using TRIC. By using the inference engine we inferred implicit relations. We checked the results of inference engine, and checked the model for consistency using the consistency checking engine. A number of iterations were needed to resolve the contradictions and to validate the inferred relations.

In the next sections we evaluate the functionality by referring to the requirements defined in Chapter 5.

8.3.1 Modeling of requirements

Described by requirements **T1 (Modeling Environment)** *and* **T2 (Modeling Requirements)**.

We specified the requirements of the CMS case in a document. The requirements are listed in tables (see Appendix A). Each requirement has an identifier, and a description. During a brainstorm session we identified the relations between requirements.

To model the requirements we entered each individual requirement using the "Add new requirement" dialog. Then we related the requirements. We stored the model to the file system and we are able to open the model again.

To be conform to the metamodel, the tool allows the specification of the status and the priority of a requirement. For the CMS case we did not take the status and priority of requirements into account. Besides storing the information in the model, the tool does not use the status or the priority during reasoning of consistency checking.

TRIC only supports one metamodel, and does not provide functionality to the user to specify or select a different metamodel.

The requirements **T1** and **T2** are fulfilled, although there is only support for one metamodel: the *core* requirements metamodel.

8.3.2 Inference and analysis of requirements relations

Described by requirements **T3 (Inferencing)** *and* **T8 (Derivation analysis)**.

After the requirements are modeled, the requirements engineer can run the Inference Engine by clicking on a button. After a few seconds the inferred relations appear in the requirements editors. The distinction between a given

relation and an inferred relation is made by different colors of the rows and a text displaying either 'given' or 'inferred'.

We verified the correctness of the Inference Engine using automated Unit tests. We created test cases for combinations of requirements relations, such as: contains(R_1, R_2) $\wedge$ refines(R_2, R_3) $\rightarrow$ requires(R_1, R_3).

By asserting the outcome (in this case 'requires(R_1, R_3)'), the Unit test will fail when the engine does not conclude this relation.

The analysis of inferred relations is supported in two ways. One way is to get a textual derivation trace in the console of the tool. Sometimes there are multiple paths to derive an inferred relation. All paths are listed. However, the textual traces are not easy to understand, no descriptions of the requirements are provided either.

The other way to get an explanation of an inferred relation is to use the Visualization Engine. A graph is displayed showing the requirements and relations involved to conclude an inferred relation. The user can click on a relation. This results in the display of the names and descriptions of both requirements. The requirements engineer can then verify whether the relation corresponds with the textual description of the requirement. While there might be multiple paths, the Visualization Engine only displays one. This path is arbitrarily chosen. For proper analysis it might be useful when all paths could be shown, or only the shortest path.

The requirements **T3** and **T8** are fulfilled by the tool.

8.3.3 Consistency checking

Described by requirements **T4 (Consistency Checking)** *and* **T7 (Reporting inconsistencies)**.

The Consistency Checking Engine requires the user to run the Inference Engine first. When the user tries to run the Consistency Checking Engine without inferencing first, the tool shown an alert and does not continue with consistency checking.

The consistency checking process takes a couple of seconds. When no inconsistencies are found, the tool will show a dialog stating that the model is consistent. When one or more consistencies are found, the inconsistencies are listed textually in the console. An example of such a report is as follows.

```
[Inconsistency]
Reason: 'Contradicting subsets of systems'
Involved requirements:
- res://Requirement/3
- res://Requirement/1
```

While the tool does detect the inconsistencies, the reporting is not very friendly for the user. When there are multiple relations between requirement 3 and requirement 1, it is not clear which *relation* caused the inconsistency.

Like the Inference Engine, we verified the correctness of the reasoner rules of the Consistency Checking Engine with unit tests.

Requirement **T4** is fulfilled and **T7** is fulfilled minimally. The tool does reports inconsistencies, but it is up to the user to pinpoint the exact requirements relation that caused the inconsistency.

8.3.4 Support for other metamodels and customizable reasoner rules

Described by requirements **T5 (Custom inference rules)** *and* **T6 (Custom consistency rules)**.

The requirements metamodel and the rules for inferencing and consistency checking are all specified in separate files. However, the tool does not support multiple metamodels and hence does not support multiple sets of rules for inferencing and consistency checking.

The rules for inferencing can consistency checker can be modified with a basic text editor. The rule syntax is straight-forward for one understanding basic first-order logic.

The requirements metamodel is defined as a OWL ontology, and can be displayed and edited using an OWL editor such as Protégé [36].

Internally the tool is prepared to work with different metamodels, the requirements relations are not hard-coded. However, without modification the tool does not support additional relationship types or other customizations of the requirements metamodel.

The requirements **T5** and **T6** are not fulfilled in the current prototype.

8.3.5 Iterative process

To support the iterative process of the requirements engineer (updating the requirements model after analyzing implicit relations and inconsistencies) was not an explicit requirement.

The implicit relations reveal more information about the modeled requirements. When inspecting the implicit inferred relations, the Requirements Engineer is able to check whether the implicit relations make sense. For example if a *requires* relation is inferred, the description of both requirements should indicate that there is indeed such a relation. If this is not the case, the requirements engineer must analyze why there is an implicit relation. A wrong inferred relation indicates that there is at least one wrong given relation.

The requirements engineer can update the model, and start the process of inferencing and consistency checking again.

8.3.6 Storage of models

In the architectural design of the tool, we described in Section 6.2 and Section 7.3.2, we mention about different *copies* of the requirements model. One model which represents the entered requirements, a copy of this model with additional information for inferencing, and a copy of the inferencing model for the consistency checking. The intention of this design was to prevent pollution of the entered model with elements only needed for reasoning (such as system instances and formula relations between requirements). However, we noticed that in some cases the model stored to the file system does contain data from the Inference Engine and the Consistency Checking Engine. The workaround is to store the model prior to running the Inference Engine.

8.4 Quality of design and implementation

In this section we evaluate the quality of the design and implementation of TRIC. We use the nonfunctional (or quality) requirements specified in Chapter 5.

8.4.1 Extensibility

According to [4] extensibility is *the property that simple changes to the design of a software artifact require a proportionally simple effort to modify its source code.*

The architectural design of TRIC is layered, and a Model-View-Controller pattern is applied to separate the system into three parts: a model, a set of controllers, and a presentation layer. This enables changes and extensions to the tool, without big changes to the code. As we will discuss in the Future Work section (Chapter 9), there are a number of possible extensions to the tool.

Change impact analysis is an example, the model remains unchanged, and the main addition will be a mechanism to track changes made to the model. Because there is a Observer-Observable pattern between the controller and the model, keeping track of changes is possible with the current architecture. Although it depends on the approach taken to realize change impact analysis.

Another point of extension is the support for other requirements metamodels. The architecture of the tool is prepared since the metamodel and the rules are defined in separate files and are loaded at run-time. However some parts in the Inference Engine are hard-coded (for instance the decomposition of the partial-refines relation) and might not apply to other metamodels.

8.4.2 Scalability

The requirement about scalability states that the performance is not an issue, but the tool should be able to handle large models. Scalability is a poorly defined, but often used term in the software engineering practice. Duboc et al. [12] define it as follows:

> "A quality of software systems characterized by the causal impact that scaling aspects of the system environment and design have on certain measured system qualities as these aspects are varied over expected operational ranges."

In TRIC the only 'scaling aspect' is the size of the requirements model. There is only one user, and only one model opened at a time. We are interested in the impact on the system qualities concerning efficiency:

Time behavior Attributes of software that bear on response and processing times and on throughput rates in performances its function.

Resource behavior Attributes of software that bear on the amount of resource used and the duration of such use in performing its function.

Both definitions are taken from the ISO/IEC 9126-1991 Standard for Software Quality Evaluation [22].

Time behavior

When observing the tool's performance, we note that the performance is sufficient for conducting case studies. The inference and consistency checking engine takes a couple of seconds to infer the implicit relations of a model containing 122 requirements and 123 relations (the model containing the initial requirements for the Course Management System).

To investigate the time behavior of TRIC with different model sizes, we took the model of the Course Management System and inserted copies of all requirements and relations. This effectively doubled the model in size. We repeated this step to create a model four times as big as the CMS model.

In Figure 8.1 the execution time of TRIC is shown for inferencing and consistency checking. The vertical axis has a logarithmic scale. The time needed for inferencing seems to be linear with respect to the size of the model. Consistency checking is taking more time with larger models, the results indicate an exponential correlation between execution time and model size.

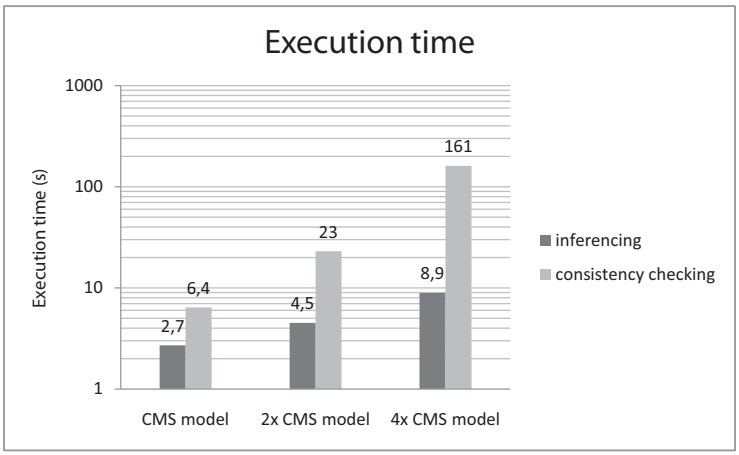

Figure 8.1: Execution time for inferencing and consistency checking

Using the Eclipse Test & Performance Tools Platform Project (TPTP) we did an Execution Time Analysis, to inspect which calls take a long time to finish. We could realize an optimization of a method in the Inference Engine that prepares the model for inference (discussed in Section 7.4.3). It turned out that the method was using a *sorted* list of requirements, while it is not necessary to have it sorted. By adding an extra parameter to the `list()` method in the Data Access Object, we could have the returned list to be sorted on request. Since sorting is a time and resource consuming operation, we reduced both time and resources needed to prepare the requirements model for inferencing.

There might be more of this possible enhancements (such as caching of frequently used model data), but we did not intend to create a maximal performing tool. However when execution time is becoming an issue, the TPTP project could be a valuable platform to search for possible improvements.

Resource behavior

The requirements model and the derived inference and consistency models are stored and manipulated in-memory during operation. Therefore, the size of the model has a direct impact on the amount of memory used by TRIC.

The amount of memory used highly depends on the number of relations in the model. Each relation will fire multiple inference rules, and because we use derivation logging, each derivation will consume memory. When using large models, the amount of memory might be a limitation.

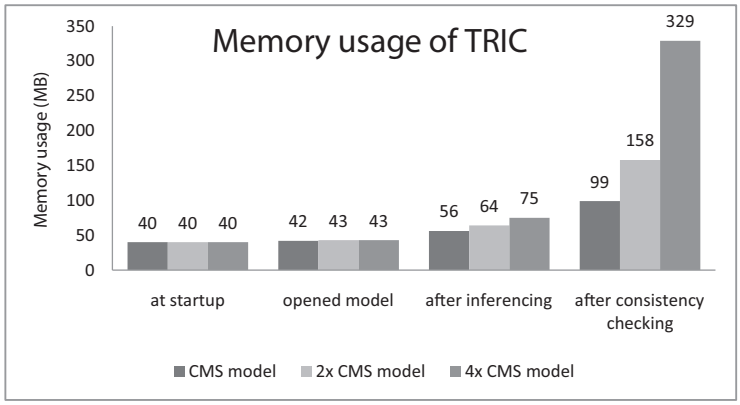

Figure 8.2: Memory usage of TRIC by activity compared to model size

We briefly investigated the memory consumption of the tool. We measured the amount of memory used for the different activities (startup, opened model, after inferencing and after consistency checking).

The result is depicted in Figure 8.2. It becomes clear that the tool consumes relatively more memory for consistency checking than for inferencing. This is because the inference model is copied to a new model, and additional information is added (formula relations). Because we use the derivation logging facility, each execution of a reasoner rule results in an entry in the inference graph.

The CMS model contains 122 requirements and 123 relations. The '4x CMS' model therefore contains 488 requirements and 492 relations. The difference between the models after inferencing is small compared to the consistency checking. Both activities indicate a linear correlation between model size and memory consumption (when we take the amount of memory at startup, 40 MB, out of consideration). More research is needed to verify this.

Using the Eclipse Test & Performance Tools Platform Project again, we did a Memory Analysis on TRIC. While the analysis did not complete correctly (the amount of memory needed for analysis exceeded the available memory), we could conclude that most memory is occupied by the JENA Framework. Every model is represented in a graph, containing nodes and triples.

For performance JENA is probably not the best choice, but as we mentioned in the design chapter, JENA is the only available reasoner offering derivation logging. This logging functionality is used extensively by TRIC to support analysis of inferred relations.

8.4.3 Interoperability

TRIC offers a platform to reason on relations in requirements models. There are a lot of existing tools outperforming TRIC with respect to functionality and usability. Also, for conducting case studies it will save time when requirements models entered with another tool could be read by TRIC.

The requirements models are expressed in RDF notation, this is an open and interchangeable data format. When third-party tools are able to store their models in XML syntax, it is relatively simple to write an XSLT transformation to convert the models to the RDF notation used by TRIC. In this way, TRIC can be used for reasoning and analysis, while other tools are used to model the requirements.

8.4.4 Portability

In general Eclipse RCP applications are platform independent. Since Java is used as programming language it should run on any machine having a Java Virtual Machine (JVM). However, for the Visualization Engine we used a Windows specific library to create a bridge between the user-interface of Eclipse (SWT) and the user-interface of JGraph (AWT). Without modification of the code for the Visualization Engine, TRIC will not function on Mac OSX or on any UNIX based OS.

8.4.5 Usability

TRIC provides a graphical user interface to create requirements models, and add requirements and relations to a model. The user interface is basic, but enables the user to model requirements in a requirements model conforming to the requirements metamodel.

The (inferred) relations between requirement are only listed for each individual requirement. No overview is provided, forcing the requirements engineer to browse al modeled requirements to look for interesting inference results. The visualization of derivation traces really helps to understand to explain how the reasoner concluded a certain relation.

The inconsistencies are listed textually in the console. They refer to requirements instead of relations. It is up to the requirements engineer to investigate which relation caused the inconsistency, which is non-trivial.

There are some functions lacking to make the tool user-friendly. For instance the tool does not provide a function to list all modeled requirements and does not give an overview of all relations inferred by the inference engine. Also, the tool does not allow the user to delete requirements.

8.5 Conclusion

We evaluated the tool using the functional and nonfunctional requirements defined in Chapter 5. To evaluate the tool support for the modeling process we created an example case for a Course Management System (CMS). The requirements for the CMS this are listed in Appendix A.

TRIC supports the modeling of requirements, requirements can be entered using a dialog. This ensures the modeled requirements conform to the requirements metamodel (i.e. each requirement has a name, description, status and priority). With another dialog requirements can be related to each other. While the tool prevents a self-relation (for example R_1 *refines* R_1), it does not check for consistency until the requirements engineer runs the consistency checker.

The tool infers the implicit relations and checks the model for consistency according to the formal semantics of the requirements relations. We verified the correctness of the reasoner with unit tests.

The tool does not support multiple requirements metamodels. The design is prepared to be extended to support this.

We investigated the scalability of the tool. The modeled CMS case is inferred and checked for consistency in seconds. Larger models take relatively more time. For consistency checking the execution time it seems there is an exponential correlation with the model size. The memory consumption for both inferencing and consistency checking seems to be linear to the size of the model, more research is needed to verify this.

Because the models are stored in an interchangeable data format (RDF/XML), it is possible to define XML transformations to and from models of other tools (if these tools also store there models using XML syntax).

The usability of the tool can be improved, though the modeling of requirements and the analysis of the implicit relations is fully supported. Inconsistencies are found, but the tool does not proved an explanation which relations caused the inconsistency.

In the next chapter we conclude this thesis and provide directions for future work.

9 Conclusion

9.1 Introduction

This chapter concludes this thesis, and gives directions for future work.

First we summarize our work in Section 9.2. In Section 9.3 we revisit the research questions. In Section 9.4 we reflect on our work and discuss the metamodeling approach for reasoning on requirements. In Section 9.5 we discuss directions for future work.

9.2 Summary

In the Software Engineering practice software requirements are one of the earliest artifacts describing a system. Without requirements we cannot verify the quality of a delivered software product. Requirements are mostly textual descriptions. Traceability is considered essential to manage consistency between software development artifacts. Most research work focused on the relation between requirements and other artifacts such as design, code and test cases. However less attention is paid to the relation *between* requirements.

Göknil et al. [19][17] proposed a requirements metamodel. This provides structure to requirements models. This metamodel is distilled from key entities from several requirements engineering approaches described in literature. The main focus of the requirements metamodel is on requirements relations and their types. Furthermore, [17] provides formal semantics of the requirements relations in first-order logic. This enables reasoning on requirements and consistency checking. The requirements metamodel and the formal semantics are described in Chapter 4.

To provide a proof of concept for the metamodeling approach for reasoning about requirements proposed by Göknil et al. [19] we need an environment to model requirements conforming to the requirements metamodel. And we need a tool that supports first-order logic reasoning over requirements relations.

To the best of our knowledge, no requirements management tools exists which are capable of reasoning about requirements relations using formal semantics. Therefore, we developed a tool named TRIC (Tool for Requirements Inference and Concistency checking). TRIC is developed as an Eclipse RCP application. In Chapter 5 we described the requirements for the tool.

Requirements models are expressed in the Web Ontology Language (OWL), because there are first-order logic reasoners for OWL. The JENA Semantic Web Framework is used as OWL API. We especially chose for the JENA framework because its reasoner provides derivation logging. We use this logging to analyze and visualize the derivation trace of inferred relations. We described the design of TRIC in Chapter 6.

To establish inference of implicit relations and to enable consistency checking we created a mapping between the formalization of requirements relations to OWL syntax and reasoner rules. Because the formal semantics are expressed in first-order logic, this mapping could be done straight-forward. Because of its more complicated formal semantics we needed a specific approach for the *partial-refines* relation. The implementation is described in Chapter 7.

We evaluated TRIC using an example case of a Course Management System. We used the requirements for the tool to verify the design and implementation. The modeling of requirements in models conforming to the requirements metamodel is supported. The inference of implicit relations and consistency checking of the model is supported. The analysis of implicit relations is supported by a visualization engine. It depicts the derivation steps of the reasoner. Inconsistencies are reported, but there is no explanation for which requirements relations are contradicting.

TRIC does not support multiple metamodels, though the design is prepared to be extended to support this. We investigated the scalability of the tool by looking at the time and resource behavior. The inferencing and consistency checking process takes a couple of seconds with the CMS case model (122 requirements, 123 relations). But for consistency checking the time increases exponentionally with larger models. The memory consumption seems linear with respect to the model size. The evaluation of the tool is described in Chapter 8.

9.3 Answering the research question

Our main research question is:

> *Can a tool be developed to support modeling of and automated reasoning on requirements models in order to infer implicit requirements relations and detect inconsistencies?*

We designed and implemented a tool capable of reasoning about requirement relations. The formalization of the requirement relations is defined for the relationship types in the requirements metamodel.

From the main question we derived a number of sub questions. We answer them separately.

> *What are the requirements for the tool?*

We identified two stakeholders for the requirements tool: metamodel engineers and requirement engineers. The first is concerned with the definition of

the requirements metamodel and the formal semantics of the requirement relationships. The requirement engineer is the actual user of the tool. He/she enters requirements into a model conforming to a selected requirements metamodel.

The tool should support the modeling of requirements conforming to a given metamodel. While the requirements metamodel is intended to be customized, the tool support we developed is only supporting one requirements metamodel. Since the metamodel and the reasoner rules are defined in separate files, the tool is prepared to be extended to support other metamodels. The tool is limited to reason using OWL-DL (Description Logics), which is a subset of first-order logic.

The tool should be capable of automated reasoning about requirements using the formalization of requirements relations in first-order logic. Contradicting (inconsistent) requirement relations should be detected by the tool.

In Chapter 5 we listed all requirements for the tool.

> *How can requirements models be represented to conform to a metamodel and to enable reasoning?*

We develop the tool using the Eclipse framework. For representing models the usage of the Eclipse Modeling Framework (EMF) would be an obvious choice. However, to the best of our knowledge there exist no first-order logic reasoners capable of reasoning directly about EMF models. Therefore we chose to represent both requirements metamodels as requirements models as Web Ontology Language (OWL) ontologies.

OWL offers a structured representation and can represent the logical property characteristics of relationships. The latter capability is very useful because a standard OWL reasoner can infer some of the implicit relations using these characteristics (in our case transitivity and symmetry).

The OWL ontology representing the requirements metamodel is imported into each requirements model. This creates an explicit link between the metamodel and the conforming model. The conformance is guaranteed by the tool implementation, though the tool is currently implemented to support only one metamodel.

> *Which reasoner engine can be used to reason on requirements models?*

There exists a number of OWL-DL (Description Logics) reasoners. We chose the JENA Semantic Web Framework as OWL API to create and manipulate the requirements models as OWL ontologies. While there are reasoners which are better performing and supporting advanced OWL reasoning, we chose for the reasoner provided by the JENA framework. The main reason is its capability to log derivation traces. During inference an internal inference graph structure is used. This graph can be inspected afterwards to analyze which facts and rules were used to conclude a certain implicit relation. This capability is of great benefit for the derivation analysis and the visualization of inferred relations. In Chapter 7 we describe how we applied this to the Visualization Engine.

Another benefit of the JENA reasoner is the support for custom reasoner rules. While there exists standards for reasoner rules (such as SWRL [20]), JENA uses its own custom rule format. The rules are easy to read and therefore easy to create and change by a metamodel engineer.

How do we create a mapping between the formalization of requirements and requirements relations to reasoner rules?

The formalization of requirements, as described by Göknil et al. [19][17], is in first-order logic. We describe the formalization in Chapter 4. The formalization is mapped to OWL-DL by defining the logical property characteristics of the requirement relationship types in the metamodel and by creating custom reasoner rules. While OWL-DL is a subset of first-order logic, no expressivity is lost with the mapping of the formalization.

How well does the resulting tool support the requirements reasoning approach?

We evaluated the tool by using the example case of the Course Management System (CMS). We modeled the requirements and their relations using TRIC and analyzed the inferred relations and inconsistencies. In a number of iterations we refined the requirements model by removing erroneous requirements relations.

The textual derivation traces and visualization of those traces were very helpful to identify the given relations that caused a certain inferred relations. For the inconsistencies there is a lack of proper explanation of the cause of an inconsistency. It would be useful to trace which *given* requirements relations are involved.

The inference and consistency checking process of the CMS case is performed in a few seconds. We briefly investigated the time behavior of the tool with larger models. A model which is four times larger takes around two minutes to complete consistency checking. Because the models are represented and manipulated in-memory, the memory consumption might be a limitation for very large requirements models.

Since TRIC is a prototype, the usability of the tool could be improved. The tool does not allow deletion of modeled requirements. Also an overview of modeled requirements and all inferred implicit relations is missing. Both features are useful for users of the tool.

Because the CMS case is an example, industrial case studies should be conducted to validate the requirements metamodel, the formal semantics of the requirements relations and our tool support.

9.4 Discussion

The requirements metamodel should be validated with empirical research. The tool can help to conduct case studies, although we think there is a big difference between a real case study compared to the example case study of the Course Management System we used. The example case was created by having the relationship types in mind. We deliberately created requirements in order to demonstrate all relationship types.

When identifying the relations between requirements we noticed that there is often a relation which is not one of the five relations defined in the metamodel. For example when requirements mention the same functionality of a system and there is no requirement describing that functionality.

Because there is no natural language processing involved in this approach, the tool cannot verify whether relationships are correctly applied. The verification is up to the requirements engineer. The tool can infer implicit relations and check for consistency, but heavily relies on the relations identified by the requirements engineer. One missed relation can make the difference between a consistent and an inconsistent model. The tool can reveal hidden relations and improve traceability but it cannot be used to prove correctness of the model.

To use TRIC a requirements engineer does not have to know the formalization details of requirements relations. The tool does not expose the user to the first-order logic used internally. The derivation log and visualization engine provide information about *which* requirement relations were used to derive an implicit relation, but it does not explain why.

The metamodel contains more elements then we have used in the tool support. We left out entities for stakeholders and test cases, and did not distinguish requirement subtypes such as functional and non-functional requirements. Also, we did not provide query and reasoning support for the status and priority of requirements. It would be useful to check only the accepted requirements for consistency and leave the proposed requirements out. Or to check the consistency of priorities: a high-priority requirement should not require a lower priority requirement.

9.5 Future work

In this section we will describe directions for future work. The developed tool supports only one requirements metamodel. Initially the tool was supposed to support multiple metamodels. We did not implement support for this, because there are some issues to be resolved first. In Section 9.5.1 we discuss the support for multiple metamodels.

The tool is capable of consistency checking. It reports whether a requirements model contains contradictions. However it does not explain which modeled relations cause an inconsistency. Section 9.5.2 we discuss the open issues for the analysis of inconsistencies.

TRIC is developed to provide a proof of concept for reasoning on requirements. It is not intended to serve as an alternative to existing requirements management tools. To conduct large case studies using TRIC it is a cumbersome task to manually enter all requirements. An integration with industrial tools would alleviate this work. In Section 9.5.3 we discuss this integration.

In this work we focused on modeling requirements without taking changes into account. Requirements evolve over time as the demands of stakeholders are changing. To trace and analyze changes between requirements models using our tool, more research is needed. In Section 9.5.4 we discuss open issues for change impact analysis.

9.5.1 Support for multiple metamodels

In practice the requirements metamodel will most likely be adapted to a specific requirements approach. Navarro et al. [33] identify this adaptation over adoptation as a critical obstacle.

The current implementation of TRIC does not support multiple requirements metamodels. It only supports the requirements metamodel proposed by Göknil et al. [19]. Initially we intended to support multiple metamodels as the 'metamodeling approach' aims at a customizable core requirements metamodel. Each of those metamodels would have different relationship types and therefore needs specific formalization rules.

In the tool design we tried to fully separate the metamodel from the application code, but some code is specific for the metamodel. For instance the decomposition of the partial-refines relation is done programmatically.

Before extending TRIC to support multiple metamodels, research should be done on *how* and to what extend the core requirements metamodel is going to be customized. Especially a closer look at the used relationship types is needed. Are the relationship types in the metamodel fixed? Or can they be replaced or deleted?

9.5.2 Analysis of inconsistencies

The current implementation is able to detect inconsistencies using the formula relations, but it cannot provide an explanation which given requirement relations are causing the inconsistency.

Formula relations are derived from requirements relations, and there are inference rules for the formula relations (listed in Section 4.5.2 in Chapter 4). Multiple requirements relations might map to the same formula relation, and there might be multiple ways to infer a contradicting formula relation. Therefore it is not trivial to determine which given requirements relation is causing a contradiction.

Ideally the tool not only lists inconsistencies, but also provides an explanation why there is an inconsistency. The requirements engineer then is supported to analyze and resolve inconsistencies. The visualization engine and the derivation trace analysis used for explaining inferred relations might be useful to achieve this.

9.5.3 Integration with industry standard tools

For conducting future case studies investigating the metamodeling approach supported by TRIC, it is very likely that requirements are already modeled in a tool or structured document. Instead of manually entering all the requirements in TRIC, it could be worthwhile to transform an existing requirements model into an OWL model TRIC can handle. This can for example be a separate transformation (e.g. an XSLT, if the source model is also expressed in XML), as depicted in Figure 9.1.

Figure 9.1: Usage of XML transformations to integrate TRIC with other tools

For tools dealing with different relationship types, the output of TRIC (implicit relations and inconsistencies) might be useful to import in the original tool.

TRIC then becomes an external reasoner and consistency checker interoperable with existing tools.

9.5.4 Change impact analysis

One of the benefits of modeling requirements is the ability to inspect relations between requirements. When a requirement is subject to change, it is very useful to determine the impact on other requirements (or more general: artifacts). Since TRIC is aware of both explicit (given) relations and implicit (inferred) relations, it should be possible to analyze the impact of a change. In other words: what will happen when a requirement will be removed or updated?

Both the given relations as the inferred relations can be helpful to narrow down the list of candidate impacted requirements. Although more research is needed on a change impact analysis approach using a reasoner.

One issue is the deletion of requirements relations because the deleted given relation might be an inferred relation as well. When rerunning the inference engine the deleted relation appears again, now as an inferred relation.

References

[1] Abma, M. Metamodels for traceability in requirements management tools (to be published). Master's thesis, University of Twente, the Netherlands, 2009.

[2] Abran, A., Moore, J. W., Bourque, P., and Dupuis, R., Eds. Guide to the Software Engineering Body of Knowledge. http://www.swebok.org/. IEEE Computer Society, 2008.

[3] Alder, G. Design and implementation of the JGraph Swing component. Available online via www.jgraph.com/documentation.html (2002).

[4] Batory, D. S., Johnson, C., MacDonald, B., and Heeder, D. v. Achieving extensibility through product-lines and domain-specific languages: A case study. In ICSR-6: Proceedings of the 6th International Conerence on Software Reuse (London, UK, 2000), Springer-Verlag, pp. 117–136.

[5] Baudry, B., Nebut, C., and Le Traon, Y. Model-driven engineering for requirements analysis. In EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (Washington, DC, USA, 2007), IEEE Computer Society.

[6] Brottier, E., Baudry, B., Le Traon, Y., Touzet, D., and Nicolas, B. Producing a global requirement model from multiple requirement specifications. In EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (Washington, DC, USA, 2007), IEEE Computer Society.

[7] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture Volume 1: A System of Patterns, 1 ed. Wiley, 1996.

[8] Cerón, R., Dueñas, J. C., Serrano, E., and Capilla, R. A metamodel for requirements engineering in system family context for software process improvement using cmmi. In PROFES (2005), F. Bomarius and S. K. Sirviö, Eds., vol. 3547 of Lecture Notes in Computer Science, Springer, pp. 173–188.

[9] Cockburn, A. Writing effective use cases. Addison-Wesley Boston, 2001.

[10] Dahlstedt, Å. Requirements Interdependencies-Moulding the State of Research into a Research Agenda. In Ninth International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ 2003), held in conjunction with CAiSE 2003 (2003), pp. 71–80.

[11] Dean, M., and Schreiber, G. OWL web ontology language reference. W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[12] Duboc, L., Rosenblum, D., and Wicks, T. A framework for characterization and analysis of software system scalability. In ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (New York, NY, USA, 2007), ACM, pp. 375–384.

[13] Duffy, D. A., Macnish, C., Mcdermid, J. A., and Morris, P. A framework for requirements analysis using automated reasoning. In CAiSe '95: Proceedings of the 7th International Conference on Advanced Information Systems Engineering (London, UK, 1995), Springer-Verlag, pp. 68–81.

[14] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency handling in multiperspective specifications. IEEE Transactions on Software Engineering 20, 8 (1994), 569–578.

[15] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., and Sebastiani, R. Reasoning with goal models. Lecture Notes in Computer Science (2002), 167–181.

[16] Giorgini, P., Mylopoulos, J., Nicchiarelli, E., and Sebastiani, R. Formal reasoning techniques for goal models. Lecture Notes in Computer Science 2800 (2003), 1–20.

[17] Göknil, A., Kurtev, I., van den Berg, K., and Veldhuis, J. W. Semantics of Trace Relations in Requirements Models for Consistency Checking and Inferencing. [Submitted] Software and Systems Modeling (2009).

[18] Göknil, A., Kurtev, I., and van den Berg, K. G. Change impact analysis based on formalization of trace relations for requirements. In ECMDA Traceability Workshop (ECMDA-TW), Berlin, Germany (Trondheim, Norway, June 2008), J. Oldevik, G. K. Olsen, T. Neple, and R. Paige, Eds., SINTEF Report, pp. 59–75.

[19] Göknil, A., Kurtev, I., and van den Berg, K. G. A metamodeling approach for reasoning about requirements. In 4th European Conference Model Driven Architecture - Foundations and Applications, ECMDA-FA 2008, Berlin, Germany (Berlin, June 2008), I. Schieferdecker and A. Hartman, Eds., vol. 5095 of Lecture Notes in Computer Science, Springer Verlag, pp. 310–325.

[20] Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., and Dean, M. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission 21 (2004).

[21] INCOSE. Requirements Management Tools Survey. `http://www.incose.org/`, January 2009.

[22] ISO/IEC 9162-1991. Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use. ISO, Geneva, Switzerland, 1991.

[23] Kent, S. Model Driven Engineering. In Proceedings of the Third International Conference on Integrated Formal Methods (2002), Springer-Verlag London, UK, pp. 286–298.

[24] Kleppe, A., Warmer, J., and Bast, W. MDA explained: the model driven architecture: practice and promise. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[25] Koch, N., and Kraus, A. Towards a common metamodel for the development of web applications. In Web Engineering. Springer, 2003, pp. 419–422.

[26] Kurtev, I. Adaptability of model transformations. PhD thesis, University of Twente, Enschede, May 2005.

[27] Lethbridge, T., and Laganière, R. Object-Oriented Software Engineering. McGraw-Hill, New York, 2005.

[28] Lopez, O., Laguna, M., and García, F. Metamodeling for requirements reuse. In Proceedings of the V Workshop em Engenharia de Requisitos WER 2002 (2002).

[29] McBride, B. Jena: A semantic web toolkit. IEEE Internet Computing 6, 6 (2002), 55–59.

[30] Miller, J., and Mukerji, J. Mda guide version 1.0.1. Tech. rep., Object Management Group (OMG), 2003.

[31] Moon, M., Yeom, K., and Chae, H. 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 (2005), 551–569.

[32] Mylopoulos, J., Chung, L., and Yu, E. From object-oriented to goal-oriented requirements analysis. Commun. ACM 42, 1 (January 1999), 31–37.

[33] Navarro, E., Letelier, P., Mocholi, J. A., and Ramos, I. A metamodeling approach for requirements specification. JOURNAL OF COMPUTER INFORMATION SYSTEMS 46, 5 (2006), 67–77.

[34] Nuseibeh, B., and Easterbrook, S. Requirements engineering: a roadmap. In ICSE '00: Proceedings of the Conference on The Future of Software Engineering (New York, NY, USA, 2000), ACM, pp. 35–46.

[35] OMG. Systems Modeling Language (SysML) Specification version 0.9. Tech. rep., Technical report, SysML Partners, 2005.

[36] Protégé. Ontology Editor, Stanford Center for Biomedical Informatics Research. `http://protege.stanford.edu/`, January 2009.

[37] Rashid, A., Moreira, A., and Araújo, J. Modularisation and composition of aspectual requirements. In AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development (New York, NY, USA, 2003), ACM, pp. 11–20.

[38] Rodrigues, O., dAvila Garcez, A., and Russo, A. Reasoning about requirements evolution using clustered belief revision. In Advances in Artificial Intelligence SBIA 2004 (2004), vol. 3171 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 41–51.

[39] Van Lamsweerde, A., Darimont, R., and Letier, E. Managing conflicts in goal-driven requirements engineering. IEEE Transactions on Software Engineering 24, 11 (1998), 908–926.

[40] Vicente-Chicote, C., Moros, B. n., and Toval, A. Remm-studio: an integrated model-driven environment for requirements specification, validation and formatting. Journal of Object Technology, Special Issue TOOLS EUROPE 2007 6, 9 (October 2007), 437–454.

[41] Zave, P. Classification of research efforts in requirements engineering. ACM Computing Surveys 29, 4 (1997), 315–321.

A

Course Management System requirements

To evaluate the functionality of the tool we created an example requirements document. In Chapter 8 we introduce this case.

The requirements are grouped by stakeholder. Functional and non-functional requirements are listed separately.

A glossary is provided which gives additional information about concepts printed in italics.

Stakeholders

We distinguish four stakeholders, each having a different role and usage of the system, and hence having different requirements.

Lecturer Uses the CMS to provide course material to students, and to manage the course's administration (such as grading and keeping track of deliverables).

Student Uses the CMS when participating courses. The CMS provides information about courses, and is used to subscribe to courses, hand in work, collaborate with other students, etc.

Administration *Bureau Onderwijs Zaken* (Office for Educational Affairs), uses the CMS to manage courses, assigning teachers to courses, and coordinating the enrolment policies. Furthermore the administration collects all kinds of statistics using the CMS, such as grade averages per course.

Maintainer Provides technical support for CMS, creating and restoring backups is one of the responsibilities.

General requirements

R1 The system shall provide *static course information*

R2 The system shall be able to store *static course information*

R3 The system shall be able to represent *static course information*

R4 The system shall provide *dynamic course information*

R5 The system shall be able to store *dynamic course information*

R6 The system shall be able to represent *dynamic course information*

R7 The system shall provide a messaging system

Requirements of Students

Functional requirements

R8 The system shall enable students to retrieve contact information of students and lecturers of subscribed courses

R9 The system shall provide the history of a course (view contents of a course over the years)

R10 The system shall provide the history of attended courses

R11 The system shall enable students to subscribe/unsubscribe to courses

R12 The system shall enable students to subscribe/unsubscribe to exams

R13 The system shall be able to provide a collaboration environment in a course (so students can share files and notes within a team)

R14 The system shall be able to let students submit textual content

R15 The system shall be able to let students upload files

R16 The System shall allow sending messages to individuals, teams or all course participants at once

R17 The system shall allow students to create teams.

R18 Teams are created by students inviting other students in the same course using the messaging system.

R19 The system shall facilitate searches in all *static information* of courses.

R20 The system shall facilitate searches within all *dynamic information* and files in a course

R21 The system shall allow students to edit their personal information

R22 The system shall allow students to change their password

R23 The system shall provide a password reset function, which resets the password and mails it to the user

R24 The system shall notify students of events (posted news messages, team invites and scheduled exams)

R25 The system shall allow students to customize the notification behaviour

R26 The system shall allow students to view course grade statistics per semester

Non-functional requirements

Privacy

R27 The system shall protect the users privacy

R28 The system shall prevent students from viewing grades of others

R29 The system shall provide a user-customizable visibility policy for the *personal information*

Availability

R30 The system shall have high availability

R31 The system shall not have unexpected downtime

R32 The system shall have downtime at most 4 hours/month

R33 The system shall have its expected downtime announced at least 48 hours in advance

R34 The system shall have downtime only during low-intensity hours

User friendliness

R35 The system will be user friendly

R36 The system shall have bilingual support (Dutch and English)

R37 The system shall have a maximum of 3 clicks to reach any content

R38 The system shall have a single login to access all content

R39 The system shall have a consistent UI (in all the views and dialogs, the UI elements behave and are placed in a similar way)

R40 The system shall have a UI which is intuitive (the behaviour of the system is according to the intuition of a standard end user)

R41 The system shall have a descriptive UI (all UI elements should have a descriptive text)

Accessibility

R42 The system shall have high accessibility

R43 The system shall be accessible by disabled (blind) users, who should be able to navigate the system and have access to all content and functionality

Security

R44 The system will be secure

R45 The system shall allow only students to change *study information* of others

Interoperability

R46 The system shall be highly interoperable

R47 The system shall provide an export to commonly used calendar formats (allowing users to import scheduled lectures into a personal calendar)

Requirements of Lecturers

Functional requirements

R48 The system shall allow lecturers to create courses

R49 The system shall allow lecturers to create entirely new courses

R50 The system shall allow lecturers to recreate a course (copied from a previous period)

R51 The system shall allow lecturers to register *assistant lecturers*

R52 The system shall allow lecturers to prepare lecture schedules (roster)

R53 The system shall allow lecturers to upload course material for lectures

R54 The system shall enable lecturers to *manage* grades (insert, update, calculate final grade)

R55 The system shall allow lecturers to specify and change the grading policy (weights of grades, determining when a student passes the course)

R56 The system shall enable lecturers to mail multiple students at once

R57 The system shall support the use of mail merge templates, to customize mass mailings (example: Dear [Firstname], your grade is: [grade]).

R58 The system shall provide a one-click function which will mail all students of the course their grades (no template should be entered or altered)

R59 The system shall allow lecturers to *manage static course information*

R60 The system shall allow lecturers to limit the number of students subscribing to a course

R61 The system shall allow lecturers to specify enrolment policies based on grade, first-come first-serve (fcfs), and department

R62 The system shall allow lecturers to specify enrolment policies based on grade

R63 The system shall allow lecturers to specify enrolment policies based on first-come first-serve

R64 The system shall allow lecturers to specify enrolment policies based on department

R65 The system shall prevent students from subscribing to a course they dont qualify for (not completing required courses, or not from the right department(s)

R66 The system shall allow lecturers to view all *personal information* (including pictures) of people in the system

R67 The system shall enable lecturers to plan meetings with students or student teams

R68 The system shall allow lecturers to *manage dynamic course information*

R69 The system shall allow lecturers to post news messages

R70 The system shall allow lecturers to *manage* the archive

R71 The system shall allow lecturers to set the visibility of archived items (enabling them to gradually expose content to students)

R72 The system shall allow only lecturers to *manage* student teams

R73 The system shall allow lecturers to enter grades for teams (so each team member will get that grade)

R74 The system shall allow only lecturers to create new teams

R75 The system shall allow lecturers to insert students into teams

R76 The system shall allow lecturers to remove students from teams

R77 The system shall allow lecturers to delete teams

R78 The system shall allow lecturers to assign *(assistant) lecturers* to teams

R79 The system shall allow lecturers to name and rename teams

R80 The system shall provide grade statistics (averages, standard deviation, per department, per year)

R81 The system shall enable lecturers to compare grade statistics with other courses

R82 The system shall allow lecturers to duplicate courses and import materials from other courses into another course, but only from their own courses

Non-functional requirements

Security

R83 The system shall allow lecturers to view the *dynamic course information* of courses given by other lecturers

R84 The system shall allow lecturers to *manage* the dynamic content visibility (visible for students and lecturers, visible for lecturers, visible to self only)

R85 The system shall allow students to view only their own grade

R86 The system shall allow lecturers to view all grades of all students in the course

Interoperability

R87 The system shall be able to import BOZ roster information into the course roster

Availability

See Students.

User friendliness

See Students.

Accessibility

See Students.

Requirements of Maintainer

Functional requirements

R88 The system shall allow maintainers to create back-ups of the entire system

R89 The system shall allow maintainers to restore partial and complete back-ups of a specific date

R90 The system shall allow maintainers to limit the size of files being uploaded by lecturers and by students

R91 The system shall allow maintainers to limit the total available space for specific courses

Non-functional requirements

Extensibility / evolvability

R92 The system shall be easily extensible and evolvable

Testability

R93 The system shall be easily testable

Scalability

R94 The system shall be scalable

Maintainability

R95 The system shall be easily maintainable

Interoperability

R96 The system shall be interoperable with secondary university systems

Requirements of the Administration

Functional requirements

R97 The system shall allow only the administration to *manage* courses

R98 The system shall allow only the administration to create new courses

R99 The system shall allow only the administration to delete courses

R100 The system shall allow only the administration to update *static course information*

R101 The system shall allow only the administration to appoint (principal) lecturers to courses

R102 The system shall allow only the administration to specify the minimum number of students for a course. If there are too little subscriptions in a semester, that course will not be given during that semester.

R103 The system shall have no maximum limit for the number of course participants ever

R104 The system shall allow only the administration to specify the course prerequisites for students

R105 The system shall allow only the administration to specify completed courses as prerequisites of a course

R106 The system shall allow only the administration to specify the department as prerequisite of a course

R107 The system shall allow the administration to retrieve all *study and personal information* of students

R108 The system shall allow the administration to retrieve all lecturer information

R109 The system shall allow the administration to enter lecturer information

R110 The system shall allow the administration to calculate grade statistics

R111 The system shall allow the administration to calculate grade statistics per year

R112 The system shall allow the administration to calculate grade statistics per student

R113 The system shall allow the administration to calculate grade statistics per course

R114 The system shall allow the administration to calculate grade statistics per department

R115 The system shall allow the administration to calculate grade statistics using combinations of the possibilities mentioned above

R116 The system shall allow the administration to calculate the number of passed students per chair per time-period

R117 The system shall allow the administration to evaluate courses through students by means of a web-survey

R118 The system shall allow the administration to manually subscribe students to courses, bypassing requirements and enrolment policies

R119 The system shall not allow users to change information which is contained and maintained by secondary university systems

R120 The system shall automatically synchronize with secondary university systems

Non-functional requirements

Availability

See Students.

User friendliness

See Students.

Interoperability

R121 The system shall be interoperable with *secondary university systems*

Extensibility

R122 The system shall allow the administration to make exceptions with regard to student enrolment to courses

Glossary

Personal Information Information about a person, such as name, address, a picture, interests, etc.

Study Information Information about a persons study progress, such as subscribed courses, grades and exam attempts.

Assistant Lecturers Lecturers who assist the principal lecturer for a course.

Static Course Information Information of a course which does not change while a course is given, but between semesters. This includes the lecturer, amount of ects and study material.

Dynamic Course Information Information of a course which changes while a course is given. This includes news messages, archived files and roster.

Secondary University Systems All university systems which are shared by different departments, such as a central address book containing all kinds of personal information

Manage Managing involves the creation, reading, updating and deleting of information.

B
Inference rules

This appendix provides the reasoner rules for the Inference Engine. The syntax is specific for the reasoner of the JENA Semantic Web Framework [29]. The rules are discussed in detail in Chapter 7.

```
# Import OWL reasoner rules
@include <OWL>.

# Declaration of prefixes
@prefix mm: <http://trese.ewi.utwente.nl/requirements.owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix inf: <inf://inference_engine/#>.

#-------------------------------------------------------
# Permeation of disjointedness.
# Not a standard rule of the JENA OWL reasoner,
# but neccessary for inferring conflicts
#-------------------------------------------------------
[subset_also_disjoint:  (?s1 inf:subSetOf ?s2)
                        (?s2 inf:disjointWith ?s3)
                    ->
                        (?s1 inf:disjointWith ?s3)]

#-------------------------------------------------------
# Map requirement relations to subset relation
# between satisfying sets of systems
#-------------------------------------------------------
[requires_to_subset:    (?r1 mm:requires ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:subSetOf ?s2)]
```

```
[refines_to_subset:     (?r1 mm:refines ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:subSetOf ?s2)]

[contains_to_subset:    (?r1 mm:contains ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:subSetOf ?s2)]

#-------------------------------------------------------
#If there is subset relation between sets of systems,
# conclude a requires relation
#-------------------------------------------------------
[subset_to_requires:    (?s1 inf:subSetOf ?s2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?r1 mm:requires ?r2)]

#-------------------------------------------------------
# If there is a conflicts relation, conclude
#   disjointness of the sets of systems
#-------------------------------------------------------
[conflict_to_disjoint:  (?r1 mm:conflicts ?r2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?s1 inf:disjointWith ?s2)
                        (?s2 inf:disjointWith ?s1)]

#-------------------------------------------------------
# If there is disjointness of the sets of systems
#  conclude a conflicts relation
#-------------------------------------------------------
[disjoint_to_conflict:  (?s1 inf:disjointWith ?s2)
                        (?s1 inf:satisfies ?r1)
                        (?s2 inf:satisfies ?r2)
                    ->
                        (?r1 mm:conflicts ?r2)]

#-------------------------------------------------------
# Rules to infer a partial refines
#-------------------------------------------------------
[temp_req_to_p_ref1: (?r1 mm:partial_refines ?r2)
                    <-
                     (?r1 mm:refines ?rt)
                     (?r2 mm:contains ?rt)
                     (?rt isTemporal 'true'^^xsd:boolean)]
```

```
[temp_req_to_p_ref2: (?r1 mm:partial_refines ?r2)
                     <-
                      (?rt mm:contains ?r1)
                      (?rt mm:refines ?r2)
                      (?rt isTemporal 'true'^^xsd:boolean)]
```

Listing B.1: Textual output given at end of the inference process

C

Consistency rules

This appendix provides the reasoner rules for the Consistency Checking Engine. The syntax is specific for the reasoner of the JENA Semantic Web Framework [29]. The rules are discussed in detail in Chapter 7.

```
# Import OWL reasoner rules
@include <OWL>.

# Declaration of prefixes
@prefix mm: <http://trese.ewi.utwente.nl/requirements.owl#>.
@prefix cons: <cons://consistency_checker/#>.
@prefix inf: <inf://inference_engine/#>.

#-------------------------------------------------------
# Map requirement relations to formula relations
#-------------------------------------------------------
[map_refines_to_formulas:  (?r1 mm:refines ?r2)
                        ->
                           (?r1 cons:all_in_whole ?r2)
                           (?r1 cons:some_implies_in ?r2)]

[map_contains_to_formulas: (?r1 mm:contains ?r2)
                        ->
                           (?r2 cons:all_in_part ?r1)
                           (?r2 cons:all_equals_in ?r1)]

[map_part_ref_to_formulas: (?r1 mm:partially_refines ?r2)
                        ->
                           (?r1 cons:all_in_part ?r2)
                           (?r1 cons:all_implies_in ?r2)]

#-------------------------------------------------------
# Properties of formula relations
#-------------------------------------------------------
```

```
[formula_rule_1:    (?p1 cons:all_in_part ?p2)
                    (?p2 cons:all_in_whole ?p3)
                ->
                    (?p1 cons:all_in_part ?p3)]

[formula_rule_2:    (?p1 cons:all_in_whole ?p2)
                    (?p2 cons:all_in_part ?p3)
                ->
                    (?p1 cons:all_in_part ?p3)]

[formula_rule_3:    (?p1 cons:some_implies_in ?p2)
                    (?p2 cons:all_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_4:    (?p1 cons:all_implies_in ?p2)
                    (?p2 cons:some_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_5:    (?p1 cons:some_implies_in ?p2)
                    (?p2 cons:all_equals_in ?p3)
                ->
                    (?p1 cons:some_implies_in ?p3)]

[formula_rule_6:    (?p1 cons:all_implies_in ?p2)
                    (?p2 cons:all_equals_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

[formula_rule_7:    (?p1 cons:all_equals_in ?p2)
                    (?p2 cons:all_implies_in ?p3)
                ->
                    (?p1 cons:all_implies_in ?p3)]

#-------------------------------------------------------
# Consistency rules.
#-------------------------------------------------------
[inconsistency_1: (?s1 inf:subSetOf ?s1)
                  (?s1 inf:satisfies ?r1)
                ->
   addInconsistency('Circular dependency',?r1)]

[inconsistency_2: (?s1 inf:subSetOf ?s2)
                  (?s2 inf:subSetOf ?s1)
                  notEqual(?s1,?s2)
                  (?s1 inf:satisfies ?r1)
                  (?s2 inf:satisfies ?r2)
                ->
   addInconsistency('Contradicting subclasses of systems',
                       ?r1,?r2)]

[inconsistency_3:  (?r1 mm:conflicts ?r2)
                   (?r1 mm:requires ?r2)
```

```
                ->
   addInconsistency('Both conflicts and depends (req.)',
                    ?r1,?r2)]

[inconsistency_4: (?r1 mm:conflicts ?r2)
                  (?r1 mm:partially_refines ?r2)
                ->
   addInconsistency('Both conflicts and depends (prt.ref.)',
                    ?r1,?r2)]

[inconsistency_5: (?r1 cons:all_in_part ?r2)
                  (?r1 cons:all_in_whole ?r2)
                ->
   addInconsistency('Requirement both part-of and whole',
                    ?r1,?r2)]

[inconsistency_6: (?r1 cons:all_equals_in ?r2)
                  (?r1 cons:all_implies_in ?r2)
                ->
   addInconsistency('all_equals_in contr. all_implies_in',
                    ?r1,?r2)]

[inconsistency_7: (?r1 cons:all_equals_in ?r2)
                  (?r1 cons:some_implies_in ?r2)
                ->
   addInconsistency('all_equals_in contr. some_implies_in',
                    ?r1,?r2)]
```

Listing C.1: Textual output given at end of the inference process