Softw Syst Model
DOI 10.1007/s10270-009-0142-3

THEME SECTION

Semantics of trace relations in requirements models for consistency checking and inferencing

Arda Goknil · Ivan Kurtev · Klaas van den Berg · Jan-Willem Veldhuis

Received: 15 January 2009 / Revised: 3 November 2009 / Accepted: 6 November 2009

Abstract Requirements traceability is the ability to relate requirements back to stakeholders and forward to corresponding design artifacts, code, and test cases. Although considerable research has been devoted to relating requirements in both forward and backward directions, less attention has been paid to relating requirements with other requirements. Relations between requirements influence a number of activities during software development such as consistency checking and change management. In most approaches and tools, there is a lack of precise definition of requirements relations. In this respect, deficient results may be produced. In this paper, we aim at formal definitions of the relation types in order to enable reasoning about requirements relations. We give a requirements metamodel with commonly used relation types. The semantics of the relations is provided with a formalization in first-order logic. We use the formalization for consistency checking of relations and for inferring new relations. A tool has been built to support both reasoning activities. We illustrate our approach in an example which shows that the formal semantics of relation types enables new relations to be inferred and contradicting relations in requirements documents to be determined. The application of requirements reasoning based on formal semantics resolves many of the deficiencies observed in other approaches. Our tool supports better understanding of dependencies between requirements.

Communicated by Prof. Richard Paige.

A. Goknil (✉) · I. Kurtev · K. van den Berg · J.-W. Veldhuis
Software Engineering Group, University of Twente,
7500 AE Enschede, The Netherlands
e-mail: goknila@ewi.utwente.nl

I. Kurtev
e-mail: kurtev@ewi.utwente.nl

K. van den Berg
e-mail: k.g.vandenberg@ewi.utwente.nl

J.-W. Veldhuis
e-mail: j.w.veldhuis@student.utwente.nl

Keywords Requirements metamodel · Requirements traceability · Inferencing · Consistency checking · Reasoning

1 Introduction

Software development has different phases (requirement analysis, architectural design, detailed design, implementation, and testing) which result in several artifacts (e.g., requirements documents, design documents). Traceability is considered crucial for establishing and maintaining consistency between these artifacts. Requirements traceability is the ability to link requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases [19]. The benefits of requirements traceability are widely acknowledged today and there are tools to record and manage trace information. Therefore, there has been a growing interest in requirements traceability in the software engineering community and industry. Despite many advances, requirements traceability remains a widely reported problem area in industry [19]. Some requirements traceability approaches aim at generating trace information automatically [12,13]. Ramesh and Jarke [39] propose that traces need to be organized according to a modeling framework. From their empirical study, they synthesized reference models comprising the most important kinds of traceability links for various development tasks. Von Knethen et al. [53] provide a survey of traceability approaches and a taxonomy of the main traceability concepts. We use their terminology to describe the scope of our research.

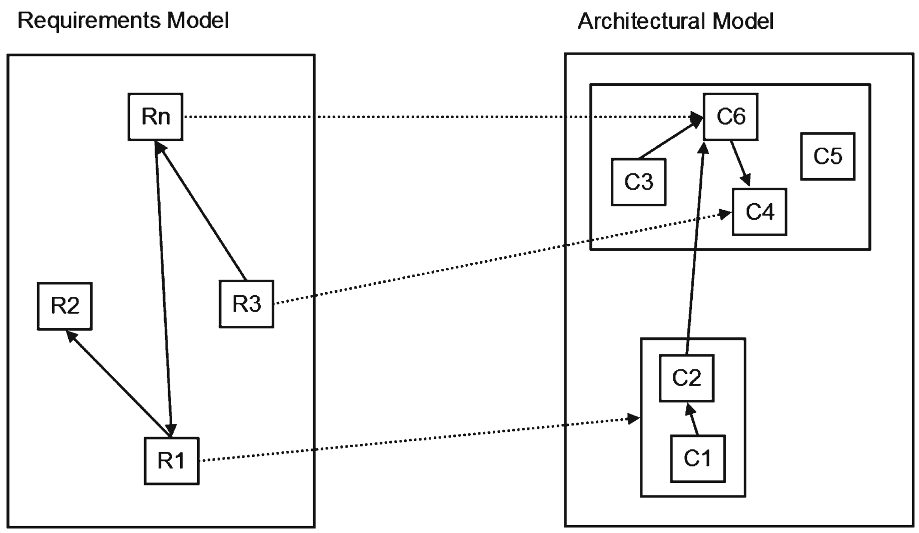


Fig. 1 Requirements model and architectural model showing within-model and between-model trace relations

1.1 Scope

Our primary interest is *post-requirements* traceability, in particular between requirements models and architectural models (see Fig. 1).

Our traceability *goal* is change impact analysis, for example, determining which model elements are impacted by a change of requirements. Therefore we need *within-model* traces and *between-model* traces. For example in Fig. 1, a change in requirement R_3 has a direct impact on architectural component C_4, and an indirect impact on component C_6 through the relation of R_3 and R_n. In this paper, we focus on relations between requirements in the requirements model (within-model requirements traceability). Any relation between requirements, between architectural components, and between requirements and architectural components can play a role in traceability analysis. Current approaches show serious deficiencies which hinder proper change impact analysis.

1.2 Deficiencies of current approaches

Considerable research has been devoted to relating requirements in both forward and backward directions. Less attention has been paid to relating requirements with other requirements. Requirements documents are considered mostly as textual artifacts with structure often not explicitly specified. In most tools and approaches, there is a lack of precise definition of requirements relations. For instance, IBM Rational RequisitePro [25] provides only two relation types between requirements: *traceFrom* and *traceTo*. In SysML [38], there are different types of requirements relations such as *contain*, *copy*, and *derive*. However, there are only informal definitions of these relations in SysML specification. Requirements relations influence a number of development activities and decisions made during software development, for example release planning, requirements validation, change impact analysis, testing, and requirements reuse [8]. In this respect, these activities may produce deficient results. For instance, change impact analysis may find impacts on nearly all requirements when a requirement is changed. A requirements engineer may have to analyze all requirements in the requirements document for a single change. This may result in neglecting the actual impact of a change. Consequently, the cost of implementing a change may become several times higher than expected. A study has shown that most requirements are related to or influence other requirements [5], and thus it is not possible to plan system releases based on only the high priority requirements without considering the relations between requirements.

1.3 Contributions

In this paper, we focus on requirements and requirements relations in requirements documents from a traceability perspective. We aim at improving requirements relations by assigning relation types and defining their semantics. Within the context of model driven engineering (MDE), we construct metamodels and models for all artifacts in software development. Therefore, we give a requirements metamodel with formal relation types. The semantics of these relations is based on first-order logic (FOL). We use this *formalization* for consistency checking of relations and inferencing. Here, *inferencing* is the activity of deducing new relations based solely on the relations which the requirements engineer has already specified. *Consistency checking* is the activity of identifying the relations whose existence causes a contradiction. We provide *tool support* for consistency checking and inferencing based on the semantics of relations for requirements. The main features of the tool are managing requirements and relations (add, update, delete), displaying consistency checking and inferencing, and explaining the results of reasoning.

1.4 Structure

The paper is structured as follows. Section 2 describes the approach. Section 3 presents the requirements metamodel and definitions of the requirements relations. Section 4 provides the formalization of the relations. In Sect. 5, we describe the use of the formalization for consistency checking and inferencing. Section 6 gives details about the tool support. Section 7 illustrates the approach by an example. Section 8 describes the related work, and Sect. 9 concludes the paper.

2 Approach

We aim at providing requirements relations with formal semantics. In order to achieve this, we successively take the following steps:

- *Requirements metamodel.* To provide an explicit structure to requirements documents, we present a requirements metamodel. This metamodel includes mostly commonly found entities in the literature. The most important elements of the requirements metamodel are requirements relations and their types (Sect. 3).
- *Semantics of relations.* Since we aim at providing requirements relations with well-defined semantics, we formalize the requirements relations in the requirements metamodel by using FOL (Sect. 4).
- *Consistency checking and inferencing.* We use the formalization for consistency checking of relations and inferring new relations (Sect. 5).
- *Tool support.* We describe the design and implementation of a prototype tool for managing requirements, displaying consistency checking and inferencing, and explaining results of reasoning (Sect. 6).
- *Running example.* We illustrate the approach with an example (Sect. 7). The example is about requirements for a course management system (CMS). This system provides a lecturer with a set of tools that allows the creation of online course content and the subsequent teaching and management of that course including interactions with students taking the course. A CMS requirements document was put together for illustration in this paper as a running example. Part of this document is given in Appendix C.

3 Requirements metamodel

Our requirements metamodel contains common entities identified in the literature for requirements models. There are several commonly used approaches to define and represent requirements: goal-oriented [34,50], aspect-driven [40], variability management [33], use-case [7], domain-specific [30,38], and reuse-driven techniques [31]. Goal-oriented requirements engineering [34,50] defines a model for decomposing a system goal into requirements with goal trees, and offers some decision methods based on this decomposition. The aspect-oriented approach [40] gives a requirements model for the separation of crosscutting functional and non-functional properties in the requirements analysis phase. The System Modeling Language (SysML) [38] is a domain-specific modeling language for system engineering. It provides modeling constructs to represent text-based requirements and relate them to other modeling elements with stereotypes. The variability management approach [33] deals with producing requirements that can be considered as a core asset in a product line.

Since we aim at using requirements relations as trace relations, we focused in our survey on the requirement entity with its attributes and relations between requirements. We left out other entities such as goals, stakeholders, and test cases. Figure 2 gives the requirements metamodel used in our approach.

In the requirements metamodel, requirements are captured in a *requirements model*. A requirements model contains *requirements* and their *relationships*. Based on Ref. [20], we define a requirement as follows.

Definition 1 *Requirement*: *A requirement* is a description of a system property or properties which need to be fulfilled.

A requirement has a unique identifier (ID), name, textual description, priority, rationale, and status. A system property can be a certain functionality or any quality attribute. In this respect, our requirements relation types and their formalization are applicable to both functional and non-functional requirements.

We identified five types of relations: *requires, refines, partially refines, contains*, and *conflicts*. In the literature, these relations are informally defined as follows.

Definition 2 *Requires relation*: A requirement R_1 *requires* a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled.

The required requirement can be seen as a pre-condition for the requiring requirement [52].

Definition 3 *Refines relation*: A requirement R_1 *refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to its properties.

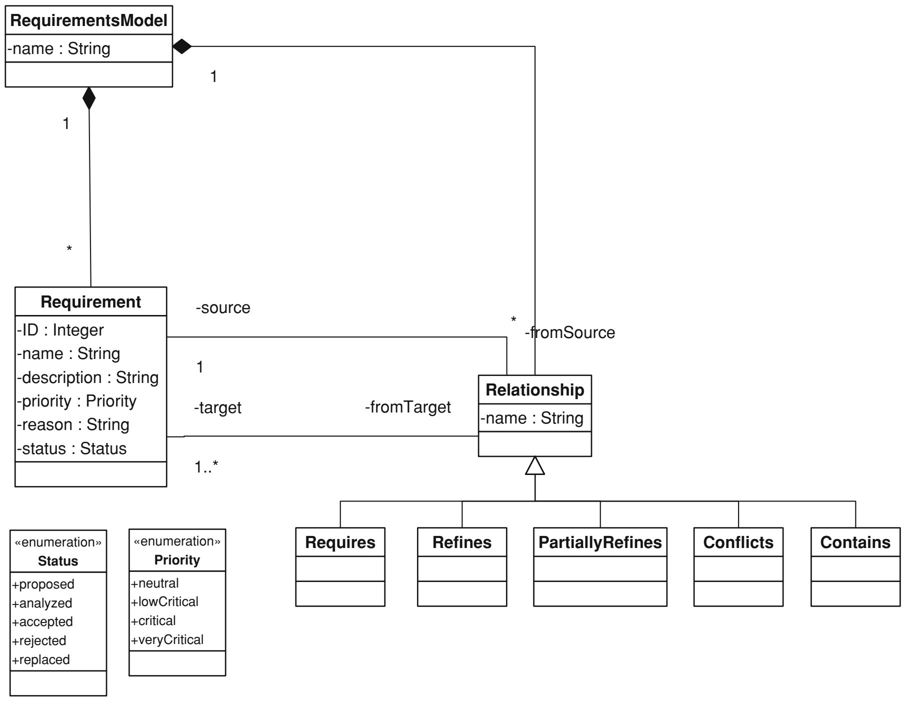


Fig. 2 Requirements metamodel

The refined requirement can be seen as an abstraction of the detailed requirements [8,50].

Definition 4 *Partially refines relation:* A requirement R_1 *partially refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to properties of R_2 and excluding the unrefined properties of R_2.

Our assumption here is that R_2 can be decomposed into other requirements and that R_1 refines a subset of these decomposed requirements. This relation can be described as a special combination of decomposition and refinement. It is mainly drawn from the decomposition of goals in goal-oriented requirements engineering [50].

Definition 5 *Contains relation*: A requirement R_1 *contains* requirements $R_2, \ldots, R_n$ if $R_2, \ldots, R_n$ are parts of the whole R_1 (part-whole hierarchy).

This relationship enables a complex requirement to be decomposed into parts [38]. A composite requirement may state that the system shall do A and B and C, which can be decomposed into the requirements that the system shall do A, the system shall do B, and the system shall do C. For this relation, all parts are required in order to fulfill the composing requirement.

Definition 6 *Conflicts relation*: 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.

The *conflicts* relation addresses a contradiction between requirements. This relation may be modeled explicitly by the requirements engineer. In this paper, we consider *conflicts* as a binary relation [49]. Our approach can be extended to n-ary conflicts relations, that is, conflicts among multiple requirements.

The conflicts relation should be distinguished from inconsistencies in requirements relations. In our terminology, an *inconsistency* is a situation where the co-existence of certain relations among requirements causes a contradiction in the context of the semantics given in this paper. When we use the term *consistency checking*, we refer to finding inconsistencies among requirements relations.

There are other classifications of inconsistencies between requirements. For example, Van Lamsweerde et al. [49] distinguish *conflicts* (excluding the simultaneous fulfillment of requirements), *divergence* (boundary cases make requirements contradict—a weaker form of conflict), *competition* (a particular case of divergence), *obstruction* (a borderline case of divergence), and *terminology clash* (using different syntactic names for a single real-world concept).

The definitions given above are informal (and sometimes ambiguous). Since we aim at precise semantics, we formalize requirements and requirements relations in FOL.

4 Formalization of requirements and relations

In this section we provide our formalization of requirements and relation types. The definitions are given in intensional and extensional terms. An intensional definition gives the meaning of a term by relating it to other terms. An extensional definition gives the set of objects that fulfill the definition.

Section 4.1 gives the formalization of requirements. Section 4.2 presents the formalization of requirements relations. In Sect. 4.3, we discuss the chosen formalization.

4.1 Formalization of requirements

We chose a formalization of requirements in FOL. In Sect. 4.3, we discuss this choice.

We assume the general notion of requirement being "a property which must be exhibited by a system". We define a requirement R as a tuple $\langle P, S\rangle$ where P is the property (or properties) and S is the set of systems that satisfy P, that is, $\forall s \in S : P(s)$. Here, P refers to an intensional definition and S refers to the extensional definition of a requirement. We formalize P as a *formula* and system s as a *model* $\mathcal{M}$ according to the model-theoretic semantics of FOL. A model $\mathcal{M}$ is the pair $(\mathcal{F}, \mathcal{P})$ where $\mathcal{F}$ is a set of function symbols and $\mathcal{P}$ is a set of predicate symbols [24]. The definition of a model in FOL is summarized in Appendix A. A satisfaction relation between the system s (captured as model $\mathcal{M}$) and the property P holds:

$$s \models_{\ell} P \tag{1}$$

where the property P computes to True in the system s with respect to the environment ℓ (i.e., a look-up table for all variables in P). P can be represented in a conjunctive normal form (CNF) in the following way:

$$P = (p_1 \wedge \cdots \wedge p_n) \tag{2}$$

where $n \geq 1$ and p_n is disjunction of literals.

A literal is an atomic formula (atom) or its negation. An atomic formula is a predicate symbol applied over terms.

From now on we assume that all formulas are in CNF. In the rest of the paper we use the notation $(p_1 \cdots p_n)$ for $(p_1 \wedge \cdots \wedge p_n)$.

Example Interpretation of a Requirement

Although the interpretation of requirements as formulas in FOL is not within the scope of our work, we give an intuition of how to map requirements in natural text to our formalization in FOL. Assume that we have the following requirement: "*The system shall provide security facilities for logging*".

We can represent the requirement as a formula *provide*$(x$, *logging*$)$ where x is a variable ranging over possible security solutions (since security can be supported in different ways, e.g., SSL certification) and *logging* is a constant. An example system s for this requirement supports SSL certification for users to log in. In the universe of concrete values, we have the value *ssl_certification*. We define the system s as a model $\mathcal{M}$ which is a pair $(\mathcal{F}, \mathcal{P})$ where:

- the non-empty set A (the universe of concrete values) contains *ssl_certification* and *logging*.
- $\mathcal{P} \stackrel{\text{def}}{=} \{\text{provide}\}$ where the concrete relation $\text{provide}^{\mathcal{M}}$ is binary.
- $\text{provide}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(\text{ssl\_certification}, \text{logging})\}$.

We have the following satisfaction relation between the system and the formula stated in the requirement:

$$s \models_{\ell} \text{provide}(x, \text{logging}) \tag{3}$$

where ℓ maps the variable x to the value *ssl_certification* in the set A and *logging* is the constant.

4.2 Formalization of requirements relations

We formalize the informal definitions of the requirements relations in the requirements metamodel.

4.2.1 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 this definition we conclude that $S_1 \subset S_2$. The subset relation between sets S_1 and S_2 defines the *requires* relation as *non-reflexive*, *non-symmetric*, and *transitive*. We disallow equality between S_1 and S_2 because the requirements engineer could put R_1 and R_2 to the same requirement when these two requirements require each other. This also excludes the reflexive property for the *requires* relation. A requirement which is a precondition for itself does not make sense in reality.

Example Requires Relation

We explain the *requires* relation with the following two requirements from the CMS requirements document explained in Sect. 7.

R24: The system shall notify students about events (new messages posted, etc.).
R7: The system shall provide a messaging facility.

In order to notify students about events like new messages posted and scheduled exams, the system needs a messaging facility. Therefore, we conclude that R24 requires R7 to be fulfilled. Please note that we consider a proper subset relation between sets of systems for these two requirements. We assume that there will always be a system which provides a messaging facility but does not provide notification to students about events.

4.2.2 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 CNF of P_2 is:

$$P_2 = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); \quad n \geq 1, \ m \geq 0 \tag{4}$$

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

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 $j \in 1 \cdots n$ such that the following two statements hold:

$$P_1 = (p'_1 \cdots p'_n) \wedge (q_1 \cdots q_m); \quad n \geq 1, \ m \geq 0 \tag{5}$$

$$\exists s \in S_2 : s \notin S_1 \tag{6}$$

From the definition we conclude that if P_1 holds for a given system s then P_2 also holds for s ($\forall s \in S_1 : s \in S_2$). On the basis of $\exists s \in S_2 : s \notin S_1$ and $\forall s \in S_1 : s \in S_2$, we conclude that ($S_1 \subset S_2$). Similarly to the previous relation we have the properties *non-reflexive*, *non-symmetric*, and *transitive* for the *refines* relation. Obviously, if R_1 *refines* R_2 then R_1 *requires* R_2.

Example Refines Relation

We explain the *refines* relation with the following two requirements.

R7: The system shall provide a messaging facility.
R16: The system shall allow messages to be sent to individuals, teams, or all course participants at once.

We formalize the requirements R7 $= \langle P_7, S_7 \rangle$ and R16 $= \langle P_{16}, S_{16} \rangle$ as follows:

$$P_7 = \text{provide\_msg}(x) \tag{7}$$

$$P_{16} = \text{course\_msg}(x) \tag{8}$$

where x is a variable over the constants *individual_msg*, *team_msg*, *participant_msg*, and *lecturer_msg*.

Let:

- $\mathcal{P} \overset{\text{def}}{=\!=}$ {provide_msg, course_msg} where the concrete relations provide_msg$^{\mathcal{M}}$ course_msg$^{\mathcal{M}}$ are unary.
- provide_msg$^{\mathcal{M}} \overset{\text{def}}{=\!=}$ {individual_msg, team_msg, participant_msg, lecturer_msg}.
- course_msg$^{\mathcal{M}} \overset{\text{def}}{=\!=}$ {individual_msg, team_msg, participant_msg}.

Then we have the following:

$$\text{course\_msg}(x) \rightarrow \text{provide\_msg}(x) \tag{9}$$

$$S_{16} \subset S_7 \tag{10}$$

R7 states only the need for a messaging property in the system. However, R16 explains the details of the messaging property: the messaging shall allow messages to be sent to individuals, teams, or all course participants at once, excluding lecturers. Therefore, we conclude that R16 refines R7. It is also noted that R16 requires R7 to be fulfilled.

4.2.3 Formalization of partially 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 CNF of P_2 is:

$$P_2 = (p_1 \cdots p_n) \wedge (q_1 \cdots q_m); \quad m, n \geq 1 \tag{11}$$

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

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

$$P_1 = (q'_1 \cdots q'_m) \tag{12}$$

$$\exists s \in S_2 : s \notin S_1, \exists s \in S_1 : s \notin S_2, \quad \text{and} \quad \exists s \in (S_1 \cap S_2) \tag{13}$$

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

Example Partially Refines Relation

We explain the *partially refines* relation with the following two requirements.

R97: The system shall allow only the administration to *manage* courses.
R102: The system shall allow only the administration to specify the minimum number of students for a course. If there are too few subscriptions in a semester, that course will not be given during that semester.

In the glossary of the CMS requirements document in Appendix C, it is stated that *managing courses* means *creating*, *updating*, *deleting*, and *reading course information*. We formalize R97 $= \langle P_{97}, S_{97} \rangle$ and R102 $= \langle P_{102}, S_{102} \rangle$ as follows:

$$P_{97} = \text{create}(x, y) \wedge \text{delete}(x, y) \wedge \text{update}(x, y) \wedge \text{read}(x, y) \tag{14}$$

$$P_{102} = \text{specify}(x, y, z) \tag{15}$$

where x is a variable for the courses, y is the variable for the number of students registered to the course, and z is the variable for the minimum number of students that should be registered to the course.

- $\mathcal{P} \overset{\text{def}}{=\!=}$ {create, delete, update, read, specify} where the concrete relations create$^{\mathcal{M}}$, delete$^{\mathcal{M}}$, update$^{\mathcal{M}}$, read$^{\mathcal{M}}$ take two arguments and specify$^{\mathcal{M}}$ takes three arguments.
- create$^{\mathcal{M}} \overset{\text{def}}{=\!=} \{(x, y) \mid x \in \text{Courses}, y \in N^+\}$, where Courses is the set of courses.
- delete$^{\mathcal{M}} \overset{\text{def}}{=\!=} \{(x, y) \mid x \in \text{Courses}, y \in N^+\}$.
- update$^{\mathcal{M}} \overset{\text{def}}{=\!=} \{(x, y) \mid x \in \text{Courses}, y \in N^+\}$.
- read$^{\mathcal{M}} \overset{\text{def}}{=\!=} \{(x, y) \mid x \in \text{Courses}, y \in N^+\}$.
- specify$^{\mathcal{M}} \overset{\text{def}}{=\!=} \{(x, y, z) \mid x \in \text{Courses}, y \text{ and } z \in N^+, y \geq z\}$.

We interpret P_{102} as assigning the minimum number of students, and the actual number of students for a given course. Then we have the following:

$$\text{specify}(x, y, z) \rightarrow \text{create}(x, y) \tag{16}$$

In this way P_{102} satisfies condition (12). For brevity we will not show any concrete model $\mathcal{M}$. Furthermore, it is easy to provide other interpretations of course creation so that $\exists s \in S_{97} : s \notin S_{102}$. Similarly, P_{102} may be satisfied by systems that do not contain, for example, the concept of course deletion and thus do not fulfill P_{97}. Therefore $\exists s \in S_{102} : s \notin S_{97}$. In the same manner we may construct a model of a system that satisfies both requirements, that is, the intersection of S_{97} and S_{102} is not empty. This reasoning satisfies the conditions of the partially refines relation.

4.2.4 Formalization of contains

Let $R_1 = \langle P_1, S_1\rangle, R_2 = \langle P_2, S_2\rangle, \ldots, R_k = \langle P_k, S_k\rangle$ be requirements where $k \geq 2$. $P_2, P_3, \ldots, P_k$ are formulas in CNF as follows:

$$P_i = (p_1^i \cdots p_{mi}^i); \quad m_i \geq 1, \quad i \in 2 \cdots k \tag{17}$$

R_1 *contains* $R_2, \ldots, R_k$ iff P_1 is derived from $P_2, P_3, \ldots, P_k$ as follows:
$P_1 = P_2 \wedge P_3 \wedge \cdots \wedge P_k \wedge P'$
where P' denotes properties that are not captured in $P_2, P_3, \ldots, P_k$.

In the definition, we do not assume completeness of the decomposition [50]. From the definition we conclude that if P_1 holds then $P_2, P_3, \ldots, P_k$ also hold, and if $P_2, P_3, \ldots, P_k$ hold then P_1 does not have to hold. Therefore, $S_1 \subset S_2, S_1 \subset S_3, \ldots$, and $S_1 \subset S_k$. Obviously, the *contains* relation is *non-reflexive*, *non-symmetric*, and *transitive.*

Example Contains Relation

We explain the *contains* relation with the following two requirements.

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.

We formalize R61 $= \langle P_{61}, S_{61}\rangle$ and R62 $= \langle P_{62}, S_{62}\rangle$ as follows

$$P_{61} = \text{allow(grade\_enrl\_policy)} \wedge \text{allow(fcfs\_enrl\_policy)} \wedge \text{allow(department\_enrl\_policy)} \tag{18}$$

$$P_{62} = \text{allow(grade\_enrl\_policy)} \tag{19}$$

where *grade_enrl_policy*, *fcfs_enrl_policy*, and *department_enrl_policy* are constants. We have the following:

$$P_{61} = P_{62} \wedge \text{allow(fcfs\_enrl\_policy)} \wedge \text{allow(department\_enrl\_policy)} \tag{20}$$

R61 states that the system shall allow lecturers to specify three different enrollment policies. The requirement can be interpreted as three different properties for the system, like *specifying enrollment policies based on grade*, *specifying enrollment policies based on first come first serve*, and *specifying enrollment policies based on department*. R62 states only one of these properties, which is *specifying enrollment policies based on grade*. Therefore, we conclude that R62 is one of the decomposed requirements of R61 (R61 contains R62). It is also noted that R61 requires R62 to be fulfilled.

4.2.5 Formalization of conflicts

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

R1 *conflicts* with R2 iff
$\neg\exists s : (s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s))$

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

Example Conflicts Relation

We explain the *conflicts* relation with the following two requirements.

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

R103: The system shall have no maximum limit on the number of course participants ever.

The satisfaction of R60 excludes the satisfaction of R103 and vice versa. The limit on the number of students and absence of a maximum limit on the number of course participants cannot exist at the same time. Therefore, we conclude that R60 conflicts with R103.

It should be noted that the definitions of *requires* and *conflicts* are given in extensional terms by specifying conditions on the corresponding sets of systems. The definitions of *refines, partially refines*, and *contains* are in intensional terms, that is, they take into account the form of the requirement specification as predicates. Our interpretation of the *refines* relation is that the refining requirement puts additional information about the same system property in the refined requirement in order to narrow down the possible solutions to the system property. The notion of adding more information about a system property is formalized by explicitly stating that p'_n implies p_n and S_1 is a subset of S_2 (see the formalization of refines where R_1 refines R_2). The intensional part of the definition states that the predicate with more information implies the predicate with less information thus resulting in a subset relation between the solutions. The subsetting captures the fact that the refining requirement limits the set of possible solutions. If only the extensional definitions were considered then we would conclude that *refines* and *requires* are equivalent, both being interpreted as a subset between the sets of systems.

4.3 Discussion of the chosen formalization

We chose a formalization of requirements and their relations in FOL. There are other formalizations of requirements, for example, in modal logic and deontic logic [32]. The formalization in FOL allows the expression of commonly occurring requirement descriptions, including for example real-time or performance requirements. However, there are limitations of the expressivity of FOL. For instance, imperfect requirements can be modeled by fuzzy sets [36]. Dealing with imperfection is out of scope of our formalization. We also do not cover modalities in requirements like possibility, probability, and necessity or logic operators like "in the next state" and "some time in the future" which can be used to describe the evolution of requirements. Our formalization should be extended with temporal logic, modal logic or fuzzy sets in order to cover these types of requirements. Under these limitations, the expressiveness of FOL is sufficient for inferencing and consistency checking since the focus of our approach is on the commonly occurring requirements.

As we stated in Sect. 4.1, the interpretation of requirements as formulas in FOL is not within the scope of our approach. The modeling of requirements and their relations is carried out by requirements engineers. However, the requirements engineer does not need to know the details of the formalization. He/she can be guided by tutorials [18] that provide an informal explanation of the relations. The requirements model is used to obtain new knowledge about the requirements relations by automated reasoning, for example, inferred relations and/or inconsistencies. These results—supported by the visualization—are presented to the requirements engineer, who should give his own interpretation. Since the requirements engineer may make mistakes in the modeling, the approach may produce improper results. However, by interpreting the results, the requirements engineer may improve his initial requirements model.

5 Inferencing and consistency checking

Inferencing and consistency checking aim at deriving new relations based on given relations and determining contradictions among relations. We provide inferencing and consistency checking that are implemented in a reasoner supporting a form of logic programming based on facts and rules. Requirements relations are represented as facts derived from their definitions. The first type of facts concerns relations among sets, and the second encodes relations between formulas. The formula relations are defined for formulas in CNF. These relations, such as *all-in-whole* and *some-implies-in*, are described in Appendix B. For example, for *some-implies-in* (P_1, P_2) at least one formula in the CNF of P_1 implies one formula in the CNF of P_2.

For example, let $R_1 = \langle P_1, S_1\rangle$ and $R_2 = \langle P_2, S_2\rangle$ be requirements and (R_1 *refines* R_2). Then, we have the following rules for the *refines* relation.

$$(S_1 \subset S_2) \text{ if } (R_1 \text{refines} R_2) \tag{21}$$

$$\text{all-in-whole}(P_1, P_2) \wedge \text{some-implies-in}(P_1, P_2) \text{ iff } (R_1 \text{refines} R_2) \tag{22}$$

The subset relation and the formula relations *all-in-whole* and *some-implies-in* are derived from the *refines* relation.

New requirements relations are derived if inferred facts are sufficient conditions for the requirements relation. According to (21) and (22), the *refines* relation is inferred if and only if the *all-in-whole* and *some-implies-in* relations are derived from the given facts.

Inferencing and consistency checking derive new facts and determine contradicting facts by using rules. The rules are generally known for set theory. We use mainly the transitive property of subset relation. The details of all formula relations and their properties can be found in Appendix B. Since the rules of set theory and formula relations can be directly mapped to Web Ontology Language (OWL) [9], we use an OWL reasoner in our implementation. The details of the tool support with OWL for inferencing and consistency checking are given in Sect. 6. In the following, we illustrate how to use the rules of set theory and formula relations in order to derive new relations and determine contradicting relations.

5.1 Inferencing

This is the activity of deriving new relations based solely on relations which the requirements engineer has already specified. The following is a proof of the inferencing that uses the set relations and their properties.

Inference: $(R_1 \text{ refines } R_2) \wedge (R_2 \text{ contains } R_3) \rightarrow (R_1 \text{ requires } R_3)$

Proof Let $(R_1 \text{ refines } R_2) \wedge (R_2 \text{ contains } R_3)$

$= \{$*By mapping from the refines and contains relations to the subset relation*$\}$

$(S_1 \subset S_2) \wedge (S_2 \subset S_3)$

$= \{$*By using the transitivity of the subset relation*$\}$

$(S_1 \subset S_3)$

$= \{$*By applying the mappings from the subset relation to the requires relation*$\}$

$(R_1 \text{ requires } R_3)$

Therefore, we have $(R_1 \text{ refines } R_2) \wedge (R_2 \text{ contains } R_3) \rightarrow (R_1 \text{ requires } R_3)$.

The reasoner is capable of automatically inferring new facts. In the implementation of inferencing, there is no need for manual proof like the one given above.

5.2 Consistency checking

This is the activity of identifying relations whose existence causes a contradiction. The following is a proof of one of the consistency checks that uses the formula relations.

Inconsistency: $(R_1 \text{ refines } R_2) \wedge (R_1 \text{ contains } R_2)$

Proof Let R_1 refines R_2.

$= \{$*By mapping the refines relation to all-in-whole and some-implies-in relations*$\}$

$(P_1 \textit{ all-in-whole} P_2) \wedge (P_1 \textit{ some-implies-in} P_2)$ **(a)**

Let R_1 contains R_2.

$= \{$*By mapping the contains relation to part-of and not-imply relations*$\}$

$(P_2 \textit{ all-in-part} P_1) \wedge (P_2 \textit{ all-equals-in} P_1)$ **(b)**

The all-in-whole relation in (a) and all-in-part relation in (b) are disjoint. They cannot exist between the same formulas together. The all-equals-in relation is symmetric and it contradicts the some-implies-in relation for the same formulas. Therefore, $(R_1 \text{ refines } R_2)$ and $(R_1 \text{ contains } R_2)$ contradict one another.

The reasoner is capable of automatically identifying contradicting facts. In the implementation of consistency checking, there is no need for manual proof like the one given above.

6 Tool support

We built a tool named Tool for Requirements Inferencing and Consistency checking (TRIC) for automatic inferencing and consistency checking [47,51]. In this section, we give the details of the tool. In Sect. 6.1, we depict the usage of the tool in the context of a modeling process. Section 6.2 gives the architecture of the tool. Section 6.3 describes the main features of the tool with some screenshots.

6.1 The modeling process

We depict the usage of the tool in a modeling process with inferencing and consistency checking. This process is based on the analysis of activities during modeling of requirements and their relations. Figure 3 gives a UML activity diagram of the process.

The process consists of the following activities.

6.1.1 Modeling

This activity takes the requirements document as input and produces the requirements model which is an instance of the requirements metamodel. The requirements model contains requirements and their relations. The definitions given in Sect. 3 are used to identify the requirements relations.

6.1.2 Inferencing and consistency checking

The modeling process is forked into two activities: *consistency checking* and *inferencing*. These two activities are processed in parallel. The requirements model is updated with inferred relations. Inconsistent parts of the model are determined, if there are any. Inferencing and consistency checking enrich the set of requirements relations in the requirements model. These two activities are in parallel because the consistency checking uses the machinery for inferencing and also checks the inconsistencies among inferred relations as well as among given relations.

6.1.3 Iterating

The process given in Fig. 3 is iterative: the requirements engineer may return to the modeling activity in order to fix inconsistencies and/or input new requirements and relations. If there is no need to update the model, the process is terminated.

6.2 Architecture

The tool is composed of three layers (see Fig. 4): (a) the *User Interface* (*UI*) *layer*, (b) the *Application Layer*, and (c) the *Data Layer*.

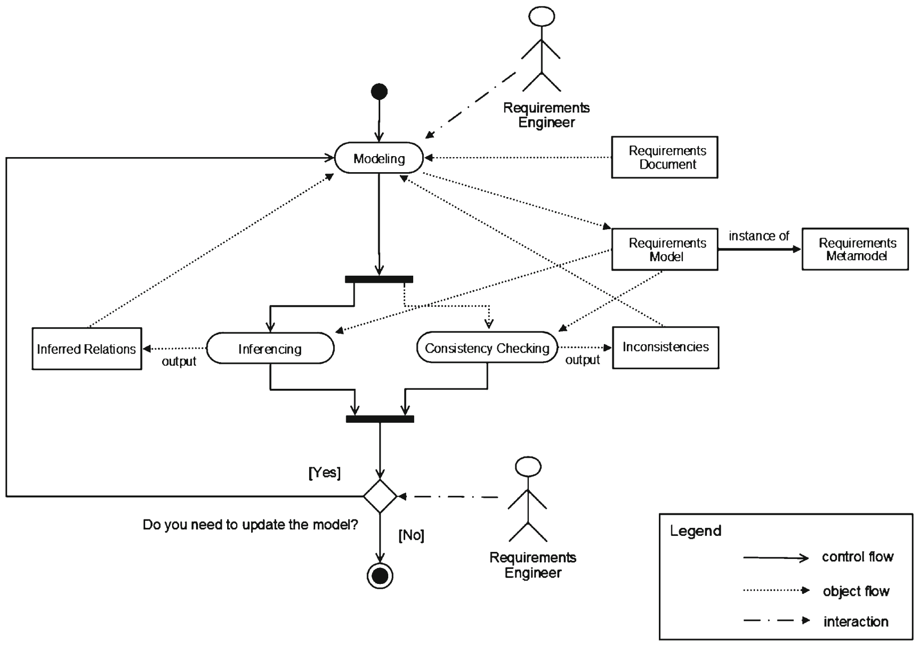


Fig. 3 Modeling process with consistency checking and inferencing

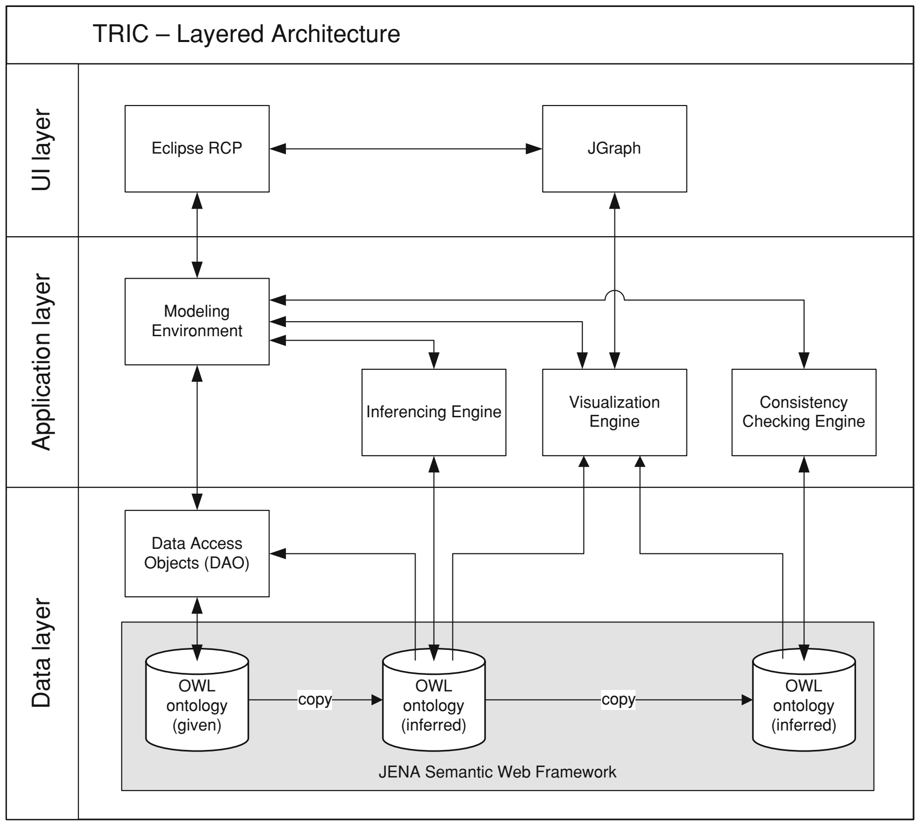


Fig. 4 Layered architecture of the tool

6.2.1 User interface (UI) layer

This layer supports the modeling activity. The user interface is implemented by using the Eclipse Rich Client Platform (RCP) [11]. The output of the consistency checking and inferencing is represented in a table form. The JGraph library [28] is used for the graphical visualization of this output. The layer provides the following:

- A form-based editor to enter and modify requirements
- An editor to enter and modify relations between requirements

- A matrix view of requirements in the model
- The control of the services provided by the application layer

6.2.2 Application layer

This layer performs the main activities given in Fig. 3: *consistency checking* and *inferencing*. It contains the components *Modeling Environment*, *Inferencing Engine*, *Consistency Checking Engine*, and *Visualization Engine*. The components provide the following functionalities:

- *Modeling Environment*: allows the creation, storage, and retrieval of requirements models, and bridging the User Interface layer with the Data layer.
- *Inferencing Engine*: infers all implicit relations between requirements, and keeps track of given and inferred relations.
- *Consistency Checking Engine*: allows checking consistency of relations.
- *Visualization Engine*: accesses the Data layer in order to get requirements and relations to be visualized in diagrams. The visualization is done by JGraph in the User Interface layer.

The *Inferencing Engine* component also implements the mappings between requirements relations and their definitions in the formalization given in Appendix B. These mappings are required to implement consistency checking and inferencing.

6.2.3 Data layer

The entered requirements and their relations are stored as an OWL ontology [9] which consists of the requirements metamodel and its instance model in the same file. Therefore, we can use the existing reasoners developed for the semantic web environment. The OWL is a family of knowledge representation languages for specifying ontologies. OWL ontologies are serialized using RDF/XML syntax. Our formalization is directly mapped to the language features of OWL like transitivity and symmetry of properties. Reasoning on requirements models is done on OWL ontologies. We used Jena [29], a programmatic environment for processing OWL data, with a rule-based inference engine. The engine performs consistency checking and inferencing. One of the advantages of Jena is that it provides derivation trace analysis. The analysis is used in one of the main facilities of the tool: *explaining results of inferencing and consistency checking*. We reason on copies of the given ontology in order to prevent the pollution of the given requirements ontology with inferred relations and inconsistencies. The Data Access Objects (DAO) component is responsible for reading and manipulating models without any dependency on data format.

6.3 Features

We describe the most important features of the tool: *managing requirements* (*add, update, delete requirements and relations*), *displaying inconsistencies and inferred relations*, and *explaining the results of reasoning*.

6.3.1 Managing requirements

We can add new requirements and update or delete existing requirements. Figure 5 gives the GUI for managing requirements which supports the modeling activity in Fig. 3.

The left-hand side of the window lists the entered requirements. The right-hand side of the window shows details of the selected requirement (R18). The tool gives a warning if a deleted given relation is inferred by the inferencing engine. The *Relate requirements* window opened by the *Add new relation*(*s*) button is used to select related requirements and relation types.

The tool provides a matrix view in order to represent and manage requirements and relations. Such a view is also available in commercial requirements management tools, such as RequisitePro. Figure 6 illustrates the matrix view feature of our tool.

The arrows with direction in the cells denote the existence of requirements relations with their directions. Since there might be multiple relations between two requirements, the tool provides the *Relate requirements* window, which is similar to the window in Fig. 5.

The matrix view is less usable for large models. We provide a visual editor (see Fig. 7) in order to improve the usability of the tool for large models. The requirements engineer can select a smaller set of requirements to be shown in a graph.

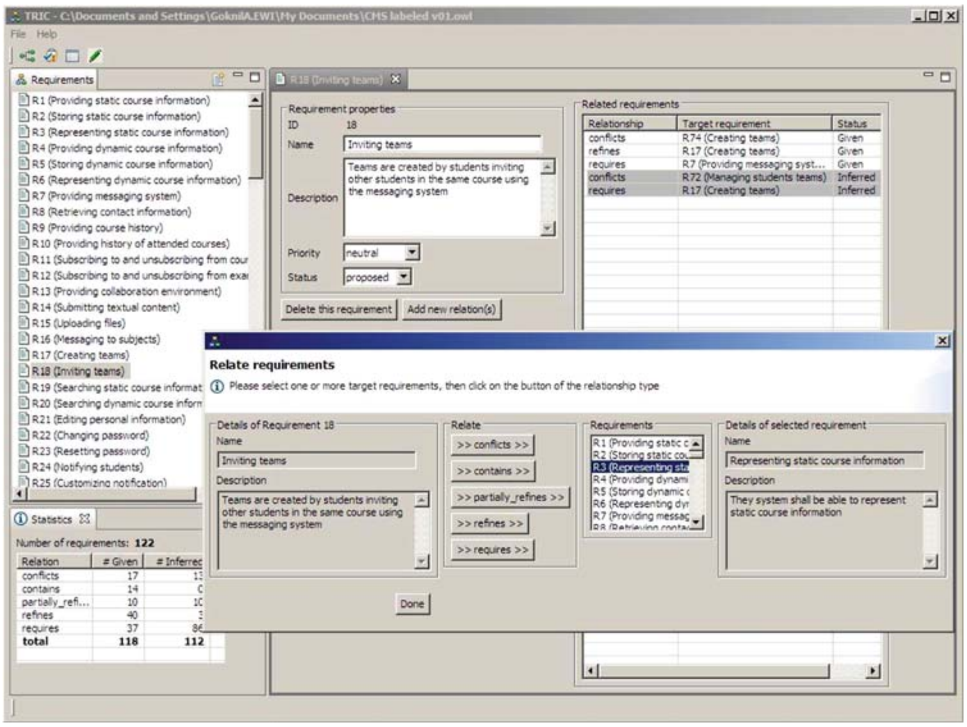
6.3.2 Displaying inconsistencies and inferred relations

The highlighted relations (*conflicts* and *requires*) in the right-hand side of the window in Fig. 5 are the inferred relations for the requirement R18. The tool detects contradictions in the model. Figure 8 gives the screenshot of output of the consistency checking activity.

The left part of the window gives descriptions of the inconsistencies; the right part gives the contradicting relations.

6.3.3 Explaining results of reasoning

The requirements engineer may need further explanation of the result from the reasoning in order to update the


TRIC - C:\Documents and Settings\GoknilA.EWI\My Documents\CMS labeled v01.owl

Requirements
- R1 (Providing static course information)
- R2 (Storing static course information)
- R3 (Representing static course information)
- R4 (Providing dynamic course information)
- R5 (Storing dynamic course information)
- R6 (Representing dynamic course information)
- R7 (Providing messaging system)
- R8 (Retrieving contact information)
- R9 (Providing course history)
- R10 (Providing history of attended courses)
- R11 (Subscribing to and unsubscribing from cour
- R12 (Subscribing to and unsubscribing from exa
- R13 (Providing collaboration environment)
- R14 (Submitting textual content)
- R15 (Uploading files)
- R16 (Messaging to subjects)
- R17 (Creating teams)
- R18 (Inviting teams)
- R19 (Searching static course informat
- R20 (Searching dynamic course inform
- R21 (Editing personal information)
- R22 (Changing password)
- R23 (Resetting password)
- R24 (Notifying students)
- R25 (Customizing notification)

Statistics

Number of requirements: **122**

Relation	# Given	# Inferred
conflicts	17	13
contains	14	0
partially_refi...	10	10
refines	40	3
requires	37	86
total	**118**	**112**

R18 (Inviting teams)

Requirement properties

ID: 18

Name: Inviting teams

Description: Teams are created by students inviting other students in the same course using the messaging system

Priority: neutral

Status: proposed

Related requirements

Relationship	Target requirement	Status
conflicts	R74 (Creating teams)	Given
refines	R17 (Creating teams)	Given
requires	R7 (Providing messaging syst...	Given
conflicts	R72 (Managing students teams)	Inferred
requires	R17 (Creating teams)	Inferred

Relate requirements

Please select one or more target requirements, then click on the button of the relationship type

Details of Requirement 18

Name: Inviting teams

Description: Teams are created by students inviting other students in the same course using the messaging system

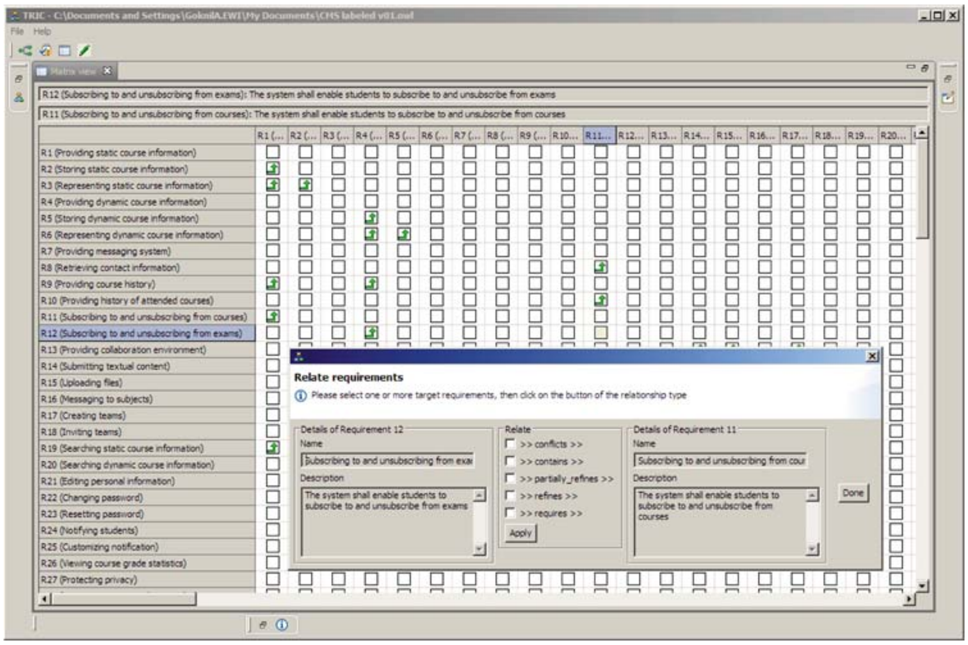
Requirements: R1 (Providing static c, R2 (Storing static cou, R3 (Representing sta, R4 (Providing dynami, R5 (Storing dynamic c, R6 (Representing dyr, R7 (Providing messag, R8 (Retrieving conta

Details of selected requirement

Name: Representing static course information

Description: They system shall be able to represent static course information

Fig. 5 GUI for managing requirements and relations


TRIC - C:\Documents and Settings\GoknilA.EWI\My Documents\CMS labeled v01.owl

R12 (Subscribing to and unsubscribing from exams): The system shall enable students to subscribe to and unsubscribe from exams

R11 (Subscribing to and unsubscribing from courses): The system shall enable students to subscribe to and unsubscribe from courses

R1 (Providing static course information), R2 (Storing static course information), R3 (Representing static course information), R4 (Providing dynamic course information), R5 (Storing dynamic course information), R6 (Representing dynamic course information), R7 (Providing messaging system), R8 (Retrieving contact information), R9 (Providing course history), R10 (Providing history of attended courses), R11 (Subscribing to and unsubscribing from courses), R12 (Subscribing to and unsubscribing from exams), R13 (Providing collaboration environment), R14 (Submitting textual content), R15 (Uploading files), R16 (Messaging to subjects), R17 (Creating teams), R18 (Inviting teams), R19 (Searching static course information), R20 (Searching dynamic course information), R21 (Editing personal information), R22 (Changing password), R23 (Resetting password), R24 (Notifying students), R25 (Customizing notification), R26 (Viewing course grade statistics), R27 (Protecting privacy)

Relate requirements

Please select one or more target requirements, then click on the button of the relationship type

Details of Requirement 12

Name: Subscribing to and unsubscribing from exa

Description: The system shall enable students to subscribe to and unsubscribe from exams

Details of Requirement 11

Name: Subscribing to and unsubscribing from cour

Description: The system shall enable students to subscribe to and unsubscribe from courses

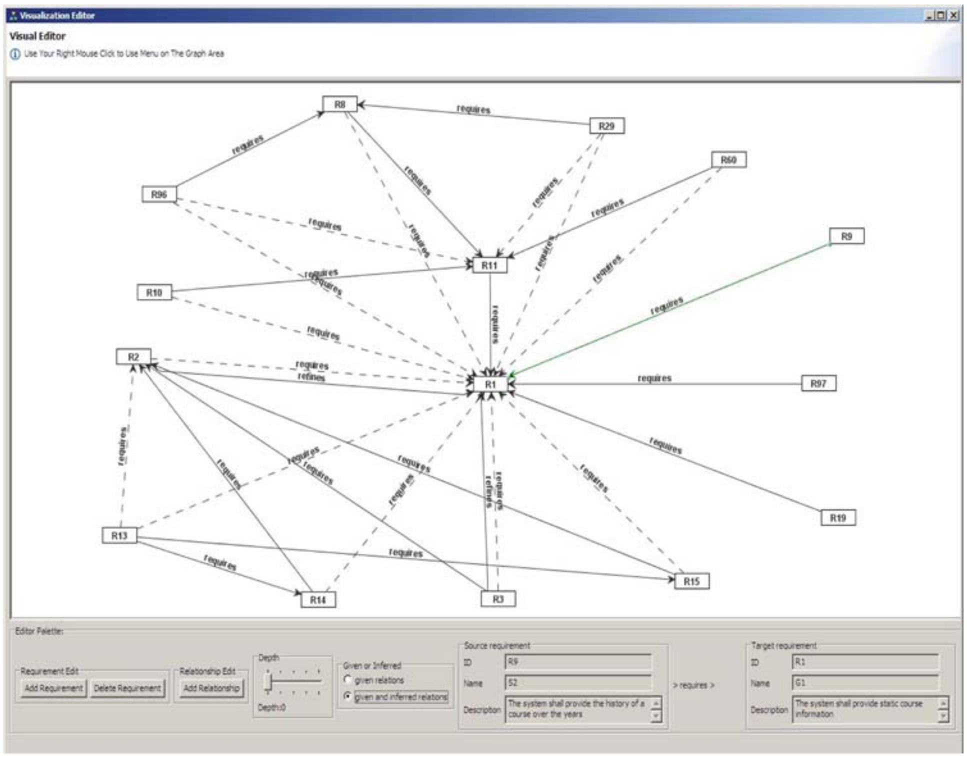
Fig. 6 Matrix view for managing requirements and relations

requirements model. The tool visualizes how inferred relations are derived (see Fig. 9).

In Fig. 9, the derivation of the *conflicts* relation between requirements R8 and R59 (dashed arrow) is visualized. Note that this conflicts relation is not an inconsistency itself. The solid arrows indicate the given relations used in the derivation.

The tool also provides an explanation of contradicting relations. This explanation is similar to the window in Fig. 9. Since the set of contradicting relations may contain inferred relations, the visualization for the derivation of inferred relations helps the requirements engineer to resolve contradictions by identifying all given relations causing the inconsistency.

Another visualization option provided by the tool is to visualize the requirements related to a selected requirement at a given depth. *Depth* is the maximum number of relations between the requirement and its related requirements in the



Visual Editor

Use Your Right Mouse Click to Use Menu on The Graph Area

Editor Palette:

Requirement Edit: Add Requirement | Delete Requirement

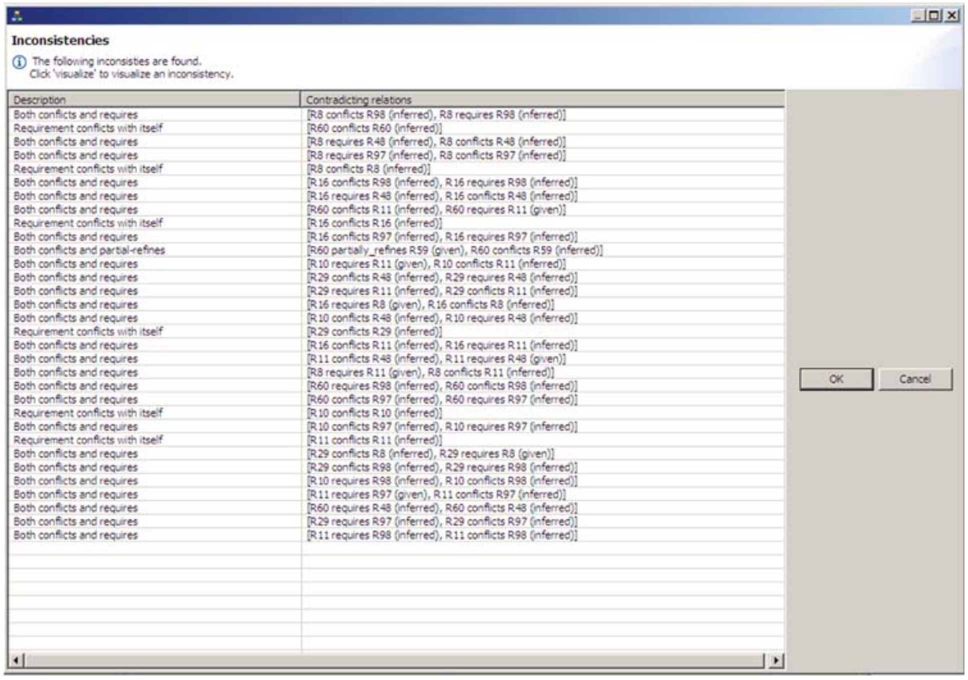
Relationship Edit: Add Relationship

Depth: Depth:0

Given or Inferred: given relations [✓ given and inferred relations]

Source requirement			Target requirement	
ID	R9	> requires >	ID	R1
Name	S2		Name	G1
Description	The system shall provide the history of a course over the years		Description	The system shall provide static course information

Fig. 7 Visual editor for managing requirements and relations



Inconsistencies

The following inconsisties are found.
Click 'visualize' to visualize an inconsistency.

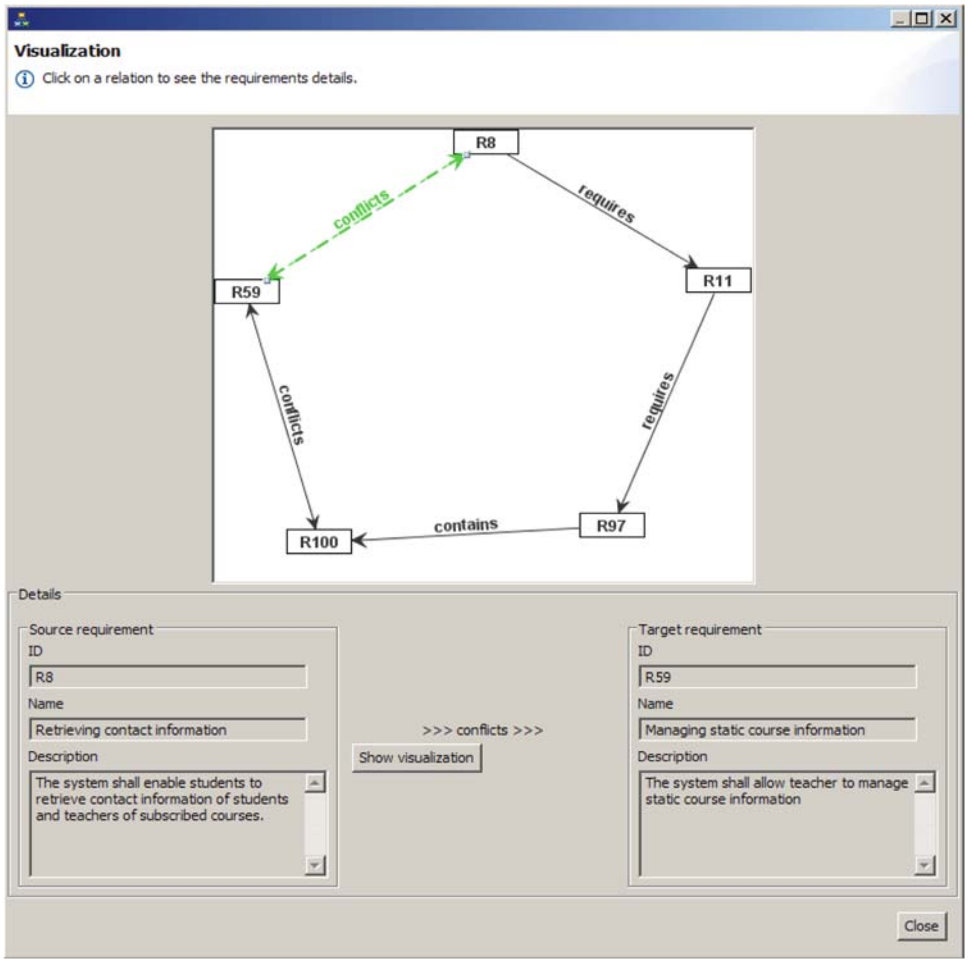
Description	Contradicting relations
Both conflicts and requires	[R8 conflicts R98 (inferred), R8 requires R98 (inferred)]
Requirement conflicts with itself	[R60 conflicts R60 (inferred)]
Both conflicts and requires	[R8 requires R48 (inferred), R8 conflicts R48 (inferred)]
Both conflicts and requires	[R8 requires R97 (inferred), R8 conflicts R97 (inferred)]
Requirement conflicts with itself	[R8 conflicts R8 (inferred)]
Both conflicts and requires	[R16 conflicts R98 (inferred), R16 requires R98 (inferred)]
Both conflicts and requires	[R16 requires R48 (inferred), R16 conflicts R48 (inferred)]
Both conflicts and requires	[R60 conflicts R11 (inferred), R60 requires R11 (given)]
Requirement conflicts with itself	[R16 conflicts R16 (inferred)]
Both conflicts and requires	[R16 conflicts R97 (inferred), R16 requires R97 (inferred)]
Both conflicts and partial-refines	[R60 partially_refines R59 (given), R60 conflicts R59 (inferred)]
Both conflicts and requires	[R10 requires R11 (given), R10 conflicts R11 (inferred)]
Both conflicts and requires	[R29 conflicts R48 (inferred), R29 requires R48 (inferred)]
Both conflicts and requires	[R29 requires R11 (inferred), R29 conflicts R11 (inferred)]
Both conflicts and requires	[R16 requires R8 (given), R16 conflicts R8 (inferred)]
Both conflicts and requires	[R10 conflicts R48 (inferred), R10 requires R48 (inferred)]
Requirement conflicts with itself	[R29 conflicts R29 (inferred)]
Both conflicts and requires	[R16 conflicts R11 (inferred), R16 requires R11 (inferred)]
Both conflicts and requires	[R11 conflicts R48 (inferred), R11 requires R48 (given)]
Both conflicts and requires	[R8 requires R11 (given), R8 conflicts R11 (inferred)]
Both conflicts and requires	[R60 requires R98 (inferred), R60 conflicts R98 (inferred)]
Both conflicts and requires	[R60 conflicts R97 (inferred), R60 requires R97 (inferred)]
Requirement conflicts with itself	[R10 conflicts R10 (inferred)]
Both conflicts and requires	[R10 conflicts R97 (inferred), R10 requires R97 (inferred)]
Requirement conflicts with itself	[R11 conflicts R11 (inferred)]
Both conflicts and requires	[R29 conflicts R8 (inferred), R29 requires R8 (given)]
Both conflicts and requires	[R29 conflicts R98 (inferred), R29 requires R98 (inferred)]
Both conflicts and requires	[R10 requires R98 (inferred), R10 conflicts R98 (inferred)]
Both conflicts and requires	[R11 requires R97 (given), R11 conflicts R97 (inferred)]
Both conflicts and requires	[R60 requires R48 (inferred), R60 conflicts R48 (inferred)]
Both conflicts and requires	[R29 requires R97 (inferred), R29 conflicts R97 (inferred)]
Both conflicts and requires	[R11 requires R98 (inferred), R11 conflicts R98 (inferred)]

Fig. 8 Output of the consistency checking activity

shortest path. The visualization is similar to the window given in Fig. 7. This visualization option allows showing only a part of the requirements model. It is useful for large models where the matrix view does not scale well.

7 Example: course management system

In this section, we illustrate our approach and tool support with the CMS example. The CMS requirements document



Source requirement: ID R8; Name: Retrieving contact information; Description: The system shall enable students to retrieve contact information of students and teachers of subscribed courses.

Target requirement: ID R59; Name: Managing static course information; Description: The system shall allow teacher to manage static course information

Fig. 9 Explanation of the inferred conflicts relation between R8 and R59

was prepared as a result of a discussion by QuadREAD project members who took the role of stakeholders. No particular inconsistencies and conflicts were inserted intentionally. We aimed at detecting inconsistencies and conflicts as a result of the modeling process. All requirements used in this paper can be found in Appendix C. We performed two iterations of the modeling process for the example.

- In the first iteration, we modeled the textual requirements and their relations according to the semantics of relation types. We analyzed given and inferred relations and inconsistencies by using the outputs of the tool. The requirements engineer identifies which relations are valid or invalid based on his or her knowledge of the application domain and the semantics of the relations. He or she decides how to correct invalid given relations by using the feature for explaining the output of reasoning.
- In the second iteration, we updated the model in order to correct the invalid relations. The validity of relations in the model was checked according to the semantics of the relation types. This checking is dependent on the requirements engineer's interpretation of the semantics of the relations.

It should be noted that the conclusions from the example cannot be generalized for our approach, since we still need to apply the approach to a number of industrial and academic case studies with empirical results. The example illustrates potential benefits and limitations of the approach for larger case studies. Section 7.1 presents the overall results of the two iterations. Section 7.2 gives some inferred relations in the example. In Sect. 7.3, we show some inconsistencies detected in the example requirements model.

7.1 Modeling the requirements

The requirements in the document are grouped by their stakeholders, which are *Student*, *Lecturer*, *System Maintainer*, and *Administration*. The functional and non-functional requirements are separated in the requirements document. There are 122 requirements (94 functional and 28 non-functional requirements). In the document, relations between requirements are not stated explicitly.

In the first iteration, we modeled the document according to our relation types by interpreting the requirements in the document. The execution of the inference engine inferred

Table 1 Number of relations and inconsistencies in the example

	Number of relations per relation type						Number of inconsistencies
	Refines	Partially refines	Requires	Contains	Conflicts	Total	
First iteration							
Given	41	9	42	14	17	123	32
Inferred	3	10	122	0	103	238	
Second iteration							
Given	40	10	37	14	17	118	0
Inferred	3	10	86	0	13	112	

new relations based on the given relations. As a second step, we run the consistency checker for the requirements model.

The tool reported 32 inconsistent parts in the requirements model. An inconsistent part is a set of relations whose existence causes a contradiction. For example, the *conflicts* and *requires* relations between R29 and R97 cause a contradiction. The output for one of the inconsistent parts is given below:

```
[Inconsistency]
Description: "Both conflicts and requires relations"
Contradicting relations:
R29 requires R97 (inferred relation)
R29 conflicts R97 (inferred relation)
```

In the second iteration, we used the tool feature for explaining the results of reasoning. The feature provides derivation trace analysis of inconsistent parts of the model. Based on this information, we discovered that there are five invalid given *requires* relations, one *refines* relation is actually a *contains* relation, and one *contains* relation is actually a *partially refines* relation in the example. Deleting and updating these relations results in a consistent requirements model. The number of inferred relations is reduced. Table 1 gives the number of given and inferred relations, and the number of inconsistencies in the first and second iteration for the CMS example.

In the first iteration there are 225 *conflicts* and *requires* relations of 238 inferred relations. Updating the model in the second iteration in order to fix the inconsistencies eliminates the inferred invalid *conflicts* and *requires* relations.

In the second iteration, reasoning on the requirements model resulted in 112 inferred relations from 118 given relations. There are 86 *requires* relations of 112 inferred relations. From the formalization of relation types, we know that the *contains* and *refines* relations imply the *requires* relation in the requirements model. Therefore, we were expecting that the number of inferred *requires* relations would be more than the total number of given *contains* and *refines* relations. Fifty-four of these 86 inferred requires relations are inferred from the given *contains* and *refines* relations. Other *requires* relations are inferred by using the transitive property of the *requires* relation and the combination of the *requires* relation with *contains* and *refines* relations.

As a result of reasoning, we have 13 inferred *conflicts* relations from 17 given *conflicts* relations. All these *conflicts* relations are inferred because of the combination of the *conflicts* relation with the *requires* and *contains* relations.

In the requirements document, the containment hierarchy has only one level. Since the transitive property of the *contains* relation is the only way to infer the *contains* relation according to its semantics, the tool does not infer any new *contains* relations. We have only three inferred *refines* relations from 40 given *refines* relations by using the transitive property of the *refines* relation. On the other hand, 10 *partially refines* relations are inferred from 10 given *partially refines* relations.

7.2 Inferring requirements relations

In this section, we describe some inferred relations in the example. The example in Fig. 10 illustrates the inferencing for the following requirements:

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

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

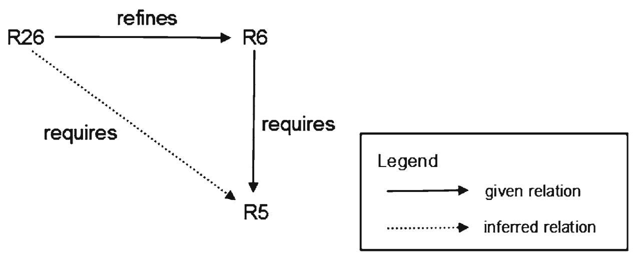


Fig. 10 Example with inferred requires relation

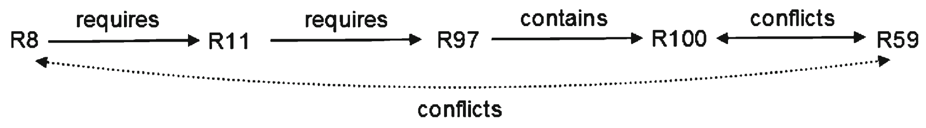


Fig. 11 Analysis of the inferred relation to identify invalid given relations

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

In the glossary of the requirements document (see Appendix C), dynamic course information is expressed as information (news messages, archived files, and roster) about a course which changes while a course is given. In the requirements model, the following relations are given: (R26 refines R6) and (R6 requires R5). When we run our tool over the requirements model, the relation (R26 requires R5) is inferred (dashed line in Fig. 10).

Grade statistics are dynamic course information. The system needs to store dynamic course information in order to allow students to view course grade statistics per semester. Therefore, we confirm that the inferred relation (R26 requires R5) is a valid relation in the model.

The interpretation of requirements depends on the requirements engineer. In the example, we discovered some invalid given relations. The tool feature for explaining the inferencing results supports our analysis of (in)valid given relations based on inferred relations. Figure 11 depicts the analysis of one inferred relation to identify invalid given relations.

Although there is an inferred conflicts relation between requirements R8 and R59, these two requirements are not in conflict. These requirements are the following:

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

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

When we analyzed the given relations used to infer conflicts relations in Fig. 11, we concluded that the given relation (R11 requires R97) is not a valid relation. These two requirements are the following:

R11: The system shall enable students to subscribe to and unsubscribe from courses.

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

R11 does not require R97 to be fulfilled. The invalid input causes the invalid output of the inferencing. The tool helps to identify candidate invalid given relations in the example.

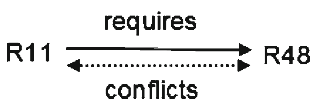


Fig. 12 Inconsistent part in the example model

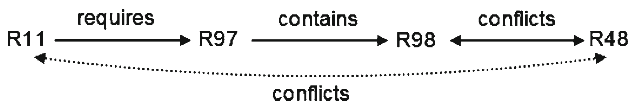


Fig. 13 Analysis of the inferred relation in the inconsistent part of the model

7.3 Checking consistency

In the previous section we treated conflicts relations, which are modeled by the requirements engineer. Here, we discuss inconsistencies, that is, contradictions among relations which are detected by our tool. We will depict how we fix an inconsistent part by using the output of our tool. The example in Fig. 12 illustrates this part. The requirements are:

R11: The system shall enable students to subscribe to and unsubscribe from courses.

R48: The system shall allow lecturers to create courses.

The consistency checking engine reports that conflicts and requires relations between R11 and R48 cause a contradiction. The relation (R11 requires R48) is a given relation. When we re-analyzed requirements R11 and R48, we concluded that this requires relation is an invalid relation. Since there might be hard coded courses in the system, the students can subscribe to and unsubscribe from these courses without any need to create courses.

Since the relation (R11 conflicts R48) is an inferred relation, we need derivation trace analysis for this relation. Figure 13 gives the analysis of the inferred relation in the inconsistent part of the model.

When we checked the given relations in Fig. 13, we found that the given relation (R11 requires R97) is an invalid relation, modeled incorrectly in the first iteration. This is the same relation we identify in the analysis of the inferred relation in Fig. 11. We removed the *requires* relations between R11 and R97, and between R11 and R48 to fix the inconsistent part in Fig. 12. This example illustrates how the tool helps localizing invalid relations.

8 Related work

We classify the related work in four categories: *Requirements Relations*, *Requirements Metamodeling*, *Requirements Reasoning*, and *Tool Support*.

8.1 Requirements relations

We studied literature about requirements relation types and their semantics. Dahlstedt and Persson [8] address requirements relations (they call a relation an "interdependency") from a traceability perspective. They give an overview of requirements relations research and present a model of fundamental relation types. There is a classification (*structural*, *constrain*, and *cost/value interdependencies*) of fundamental interdependency types which includes some of the relations (*refines*, *requires*, and *conflicts*) we also use in our approach. The need to understand the nature of requirements relations and their influence on software development activities such as change management are stated. However, there is no formal semantics for the relations. Carlshamre et al. [5] run an industrial survey of requirements in software product release planning. Their aim is to learn about the nature of interdependencies in general, to be able to classify them, and to assess the relative frequency of different classes. The results show that roughly 20% of the requirements are responsible for 75% of the interdependencies and only a few requirements are singular. It is expected to find conflicting requirements in the survey, since this relation is common in the literature. However, no such dependencies are identified. Apparently conflicts had already been eliminated in the documents.

Although the two studies mentioned above motivate the need for requirements relations, no much attention is paid for how to give formal semantics of the relations. Aizenbud-Reshef et al. [1] state the need for semantics of traceability links in general. They present an approach to defining operational semantics for traceability in UML which captures more precisely the intended meaning of various types of traceability. The main goal is achieving automated consistency management of UML models. A specific type of operational semantics for traceability in UML is described. The semantic property of a traceability relationship is a triplet (*event*, *condition*, and *actions*). This triplet is very much dependent on change impact analysis. For instance, an event indicates a change in a model. Conditions help to differentiate between events. Actions describe what should and should not be done when a specific event has occurred. Therefore, it is hard to use the semantics in [1] for other purposes like inferencing and consistency checking of trace relations. On the other hand, the semantics formalized with FOL in our paper can be considered as more general and suitable for different purposes. In [16,17], we use our semantics for inferencing, consistency checking, and change impact analysis in requirements models.

The survey in [42] introduces Requirements Interaction Management (RIM), which is concerned with the analysis and management of dependencies among the requirements. One of the activities in RIM, is reasoning on requirements interactions. Conflict detection methods for reasoning are introduced in five categories: *domain model*, *theorem model*, *scenario analysis*, *modeling checking* and *executing monitoring* methods. We consider our work in the scope of the domain model method. The domain model method is summarized in the survey that a domain model of system requirements interactions is used to identify interactions at the requirement level. We consider that our requirements metamodel is our domain model of requirements relations which stand for requirements interactions to identify relations between requirements.

8.2 Requirements metamodeling

A number of approaches in MDE address modeling requirements and their relations from a traceability perspective. Vicente-Chicote et al. [52] describe a requirements metamodel and a modeling environment. The environment supports: graphical requirements models, their validation against the metamodel and against a set of constraints written in Object Constraint Language (OCL), and automatic generation of a navigable Software Requirements Specification document (SRS). In the requirements metamodel, there are three types of trace links between requirements: *DependenceTrace*, *InfluenceTrace*, and *ParentChildTrace*. The relations are defined informally.

Baudry et al. [2] introduce a metamodel for requirements and present how they use it on top of a constrained natural language for requirements definitions. The requirements metamodel captures functional requirements as use cases with pre-conditions and post-conditions that constrain the activation of use cases. Operations are added in the metamodel in order to simulate requirements models. The goal of the simulation is to instantiate the use cases, replacing the formal parameters with actual values defined in an initial configuration. The metamodel does not capture the static part of requirements. It does not have the notion of requirements relations. On the other hand, our approach covers the static aspects of requirements including non-functional requirements and reasoning on requirements relations. In [3], a model-driven mechanism is proposed to merge different requirement specifications and reveal inconsistencies between them by using a requirements metamodel. The requirements metamodel is mainly used to produce a requirements model from a given requirements document. Requirements relations are not typed and lack semantics. Consistency checking and inferencing for requirements relations are not supported.

Some authors [21,45] use the UML profiling mechanism in a goal-oriented requirements engineering approach. Heaven et al. [21] introduce a profile that allows the KAOS model [50] to be represented in UML. They also provide an integration of requirements models with lower level design models. Supakkul et al. [45] use the UML profiling mechanism to provide an integrated modeling language for functional and non-functional requirements that are mostly specified by using different notations. These two works aim at a metamodel for goal-oriented requirements engineering rather than reasoning over requirements.

SysML [38,44] uses the UML profiling mechanism to provide modeling constructs that represent text-based requirements and relate them to other modeling elements. The relation types for requirements in SysML are *derive*, *copy*, and *contain*. SysML also provides a stereotype mechanism that allows the requirements engineer to specify their own relation types. The main goal of SysML requirements diagrams is to represent the requirements and their relations. Formal semantics of relation types is not considered. The definitions of the relations tend to be ambiguous. No reasoning facility for requirements is provided.

Navarro et al. [35] propose a customization approach for requirements metamodels. They propose a core requirements metamodel which is generic and considers only *Artifact* and *Dependency* as core entities. The metamodel does not contain concrete types for requirements relations. This disallows the application of inference rules for the core relations to customized entities. The Requirements Interchange Format (RIF) [41] structures requirements and their attributes, types, access permissions, and relationships. It is defined as an XML schema. Its data model has generic entities and relations like *Information Type*, *Association*, and *Generalization*. These entities can be formalized to reason about requirements and their relations. Ramesh et al. [39] propose reference models for requirements traceability. The models include basic entities like Stakeholder, Object, and Source. Relations between different software artifacts and requirements are captured.

Some papers address domain-specific requirements models. Koch et al. [30] propose a requirements metamodel specialized for Web systems. They identify the general structure of Web systems in order to define the requirements metamodel. Moon et al. [33] propose a methodology for producing requirements that can be considered as a core asset in the product line. Ceron et al. [6] discuss requirements modeling in the context of product lines. They propose a metamodel for requirements that contains both the common and variable parts. Lopez et al. [31] propose a metamodel for requirements reuse as a conceptual schema to integrate semiformal requirement diagrams into a reuse strategy. The requirements metamodel is used to integrate different abstraction levels for requirements definitions. All these domain-specific approaches aim at providing a structure for representing requirements and their relations. Some of them do not contain types of requirements relations and most of them only provide informal definitions of their relations.

8.3 Requirements reasoning

A number of approaches describe reasoning about requirements. Giorgini et al. [15] propose a formal framework for reasoning with goal models. A precise semantics is given for all goal relationships in a qualitative and numerical form. Label propagation algorithms that are shown to be sound and complete with respect to the axiomatization are introduced. Two main limitations are stated. One concerns the definition of contribution links and the labels assignment; the second is that the conflicts relation is not resolved. In general, the idea in [15] is similar to our approach. However, the presented reasoning framework is very specific to goal models. No reasoning facility and tool support is introduced.

Zowghi et al. [54,55] propose a logical framework for modeling and reasoning about the evolution of requirements. They characterize the properties correctness, completeness, and consistency of requirements in an evolutionary framework. The interaction of consistency and completeness with correctness during requirements evolution is discussed. Duffy et al. [10] propose a logic-based framework for reasoning about requirements specifications based on goal-tree structures. The framework is based on goal decomposition supported by automated reasoning. Rodrigues et al. [43] propose a framework for the analysis of evolving specifications that enables reasoning in the presence of inconsistency. The work is complementary to our formalization since a tool that translates requirements given in the form of “if then else” rules into the disjunctive normal form for classical logic reasoning and cluster prioritization is provided.

Heitmeyer et al. [22] propose consistency checking in requirements specifications for automatic detection of errors, such as type errors, non-determinism, missing cases, and circular definitions. The technique is based on requirements specifications expressed in the SCR (Software Cost Reduction) tabular notation. A formal requirements model that represents the system to be built as a finite-state automaton is provided. It defines a system state in terms of entities, a condition as a predicate on the system state, and an input event as a change which triggers a new system state. There are some consistency checks derived from the formal requirements model such as type correctness. Contrary to our approach, the formal requirements model requires modeling requirements in a very formal way in order to detect inconsistencies. The main focus is determining inconsistencies among requirements instead of inconsistencies among requirements relations.

Finkelstein et al. [14,37] describe a technique for inconsistency handling in requirements documents developed using multiple methods and notations for the same system. They combine the ViewPoints framework for perspective development and a logic-based approach to inconsistency handling. Partial specification knowledge in each ViewPoint is translated into FOL. Logical inconsistencies are identified. Then, some temporal logic rules are combined with the identified inconsistencies to specify inconsistency handling actions. Hunter et al. [23] present an adaptation of classical logic, which they term quasi-classical (QC) logic that allows reasoning in the presence of inconsistency. This facilitates an analysis of inconsistent information. In our approach, inconsistencies are explained based on the derivation trace of relations.

8.4 Tool support

Some requirements management tools support multiple requirements relation types. The INCOSE management tool survey [27] evaluates these tools according to the criterion traceability analysis, that is, what kinds of trace links the tools provide and what kinds of analyses are performed by the tools. According to the responses of tool vendors in the survey, current industrial tools mostly do not support reasoning about requirements relations.

IBM Rational RequisitePro [25] provides only two relation types between requirements: *traceFrom* and *traceTo*. Since these two relations indicate only the direction, they are very generic relations. In IBM Telelogic Doors [26], there is no predefined requirements relation. The requirements engineer can specify his or her own relation type. However, it is not possible to assign semantics to relation types created by the requirements engineer. The tool provides basic support for change impact analysis. It shows suspected relations when a requirement is updated. Borland Caliber [4] provides only one generic relation type for requirements. This type can be used for different purposes such as part-whole and refinement. The reasoning facilities of the tools IBM Rational RequisitePro, IBM Telelogic Doors, and Borland Caliber are based only on the transitivity property of the relations. These tools do not support consistency checking of the relations.

In TopTeam Analyst [48], there are four relation types. Three of these relations (*traces into*, *impact*, *used in*) are directed and one of the relations (*trace*) is undirected. This undirected relation is considered as a generic relation type for the other relation types. None of these relation types have formal semantics. The tool does not support any reasoning.

We may conclude that some common industrial requirements tools do not support reasoning about relations between requirements or provide formal semantics for relation types.

9 Conclusions and future work

There has recently been a growing interest in requirements traceability in the software engineering community and industry. Although considerable research has been devoted to linking requirements in both forward and backward directions, less attention has been paid to linking requirements with other requirements. In this paper, we focused on requirements and requirements relations from a traceability perspective. A requirements metamodel including relation types with formal semantics was proposed. Existing requirements engineering approaches were surveyed in order to extract the metamodel. We provided semantics of trace relations with formalization in FOL. The formalization of relations was used in tool support for inferencing and consistency checking. We illustrated the approach and tool support in the CMS requirements document.

The usage of the formal semantics of relation types enables new relations to be inferred and contradicting relations to be determined in requirements documents. It overcomes many of the deficiencies of other approaches. Our tool supports a better understanding of dependencies among requirements.

There are still open issues. In some cases, relations do not cause any contradiction but violate some of the constraints in the requirements engineering domain such as “every non-functional requirement should be related with at least one functional requirement”. These constraints may be valid only for a specific requirements engineering approach like goal-oriented requirements engineering. We plan to use OCL in order to specify these kinds of constraints in the requirements metamodel. However, we need further research to specify these constraints. Apart from specifying constraints, there might be updates in the requirements metamodel. In the formalization of relations, we stated that the *refines* and *contains* relations imply the *requires* relation. This might be interpreted as a specialization relation between the *requires*, *refines*, and *contains* relations.

Our approach uses the semantic web technologies OWL and Jena instead of MDE technologies such as model transformation languages and engines. OWL and Jena directly support inferencing by using basic properties like symmetry and transitivity. However, in model transformation languages, we have to encode all basic properties and the logic behind them in order to have the same inferencing capability.

Our current support is for textual requirements. We do not have any support for other requirements artifacts like use case or activity diagrams. We improved the usability of the tool for large models with the visual editor which enables selecting requirements to be shown. However, there is still a need to test the tool in large requirements documents. We plan to conduct an experiment and do a case study as a future work.

In our previous work [16], we presented an approach to formalizing requirements relations and reusing the formalization for customization of the requirements metamodel. The main focus of the previous work is to customize the core requirements metamodel and to apply the inference rules written for the core relations to the customized relations. We showed how we could benefit from this approach by applying it to current requirements modeling approaches like SysML. Our tool needs to be extended for this customization.

The requirements attributes like priority and status can be included in our reasoning engine. For instance, we may define the constraint that a requirement cannot require another requirement whose priority is lower.

In [17,46], we presented an approach for using requirements relations and their semantics for change impact analysis. There is still the need for further research and tool support in order to apply semantics of relations in change impact analysis. For the evolution of requirements, we want to analyze the impact of requirements changes on architectural and detailed design. The next step is to define trace relations and their semantics in order to link requirements models to design models with a similar approach presented in the paper.

Acknowledgment The research presented in this paper is part of the QuadREAD project (http://quadread.ewi.utwente.nl). The authors would like to thank Yigit Caliskan for improving the features of the tool.

Appendix A: Definition of a model in FOL

In this appendix, we recapture the terminology for defining a model in FOL [24]. A model $\mathcal{M}$ is a pair $(\mathcal{F}, \mathcal{P})$ where $\mathcal{F}$ is a set of function symbols and $\mathcal{P}$ is a set of predicate symbols. It consists of the following set of data:

- a non-empty set A, the *universe of concrete values*
- for each $f \in \mathcal{F}$ with n arguments, a concrete function $f^{\mathcal{M}}$ is defined

  $f^{\mathcal{M}} : A^n \to A$

  from A^n, the set of n-tuples over A, to A
- for each $P \in \mathcal{P}$ with n arguments, a subset $P^{\mathcal{M}} \subseteq A^n$ of n-tuples over A.

The condensed definition of *formula* in FOL using Backus Naur Form (BNF) is the following:

$$\phi ::= K\,(t_1, t_2, \ldots, t_n)\ |(\neg\phi)\,|(\phi \wedge \phi)\,|(\phi \vee \phi)\,| \\ (\phi \to \phi)\,|(\forall x \phi)\,|(\exists x \phi) \tag{23}$$

In Eq. (23), K is a predicate of arity n, t_i are terms, and x is a variable. Each occurrence of ϕ on the right-hand side of the ::= stands for any formula. A formula is in CNF if it is a conjunction of formulas, where these formulas are atomic formulas or disjunctions of other formulas (clauses). An atomic formula is a formula with no deeper structure, that is, a formula that contains no logical connectives and has no sub-formulas. The satisfaction relation between a model and a formula is the following:

$$\mathcal{M} \models_{\ell} \phi\text{, for each logical formula } \phi \text{ over the pair}(\mathcal{F}, \mathcal{P}). \tag{24}$$

This denotation says that ϕ computes to True in the model $\mathcal{M}$ with respect to the environment ℓ, a look-up table which associates with every variable x a value $\ell(x)$ of the model (ℓ : var $\to A$).

Appendix B: Mapping requirements relations to extensional and intensional definitions

Based on the extensional definitions of the relations, we can map the requirements relations (*requires*, *refines*, *contains*, and *conflicts*) to the set theoretic relations for the set of systems.

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

- $(S_1 \subset S_2)$ iff (R_1 requires R_2)
- $(S_1 \subset S_2)$ if (R_1 refines R_2)
- $(S_1 \subset S_2)$ if (R_1 contains R_2)
- $((S_1 \cap S_2) = \emptyset)$ iff (R_1 conflicts R_2)

To map the partially refines relation to the set theoretic relations for a set of systems, we decompose this relation to the combination of contains and refines relations.

Therefore, we define a temporary requirement named R_{T12} which is a tuple $\langle P_{T12}, S_{T12}\rangle$ to decompose the partially refines relation between $R_1$and R_2 into refines and contains relations. We decompose the partially refines relation into contains and refines relations in two different combinations:

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

The combinations given above can exist at the same time. Based on the intensional definitions of the relations, we can map the requirements relations (*contains*, *refines*, and *partially refines*) to the relations between the properties (P) satisfied by the system. First we define the relations (*all-in-part*, *all-in-whole*, *some-implies-in*, *all-implies-in*, *all-equals-in*) between properties. We define a model M in order to have the relations between properties.

Let $\mathcal{F}$ be a set of function symbols and $\mathcal{P}$ a set of predicate symbols, each symbol having a fixed number of required arguments. $\mathcal{F} \overset{\text{def}}{=} \{||, \text{set}, f\}$ and $\mathcal{P} \overset{\text{def}}{=} \{\text{eval, implies, related, all-in-part, all-in-whole, some-implies-in, all-implies-in, all-equals-in}, =, <\}$. Our model $\mathcal{M}$ is the pair $(\mathcal{F}, \mathcal{P})$ with a non-empty set $\boldsymbol{A}$ (the universe of concrete values). For the model $\mathcal{M}$ the following holds:

- The set $\boldsymbol{A}$ is the set of all formulas in FOL and real numbers.
- The functions $||^{\mathcal{M}}$, $\text{set}^{\mathcal{M}}$ take one formula in $\boldsymbol{A}$ as argument and return the number of clauses and the set of clauses in the CNF of the formula, respectively.
- The predicates $\text{implies}^{\mathcal{M}}$, $\text{related}^{\mathcal{M}}$, $\text{all-in-part}^{\mathcal{M}}$, $\text{all-in-whole}^{\mathcal{M}}$, $\text{some-implies-in}^{\mathcal{M}}$, $\text{all-implies-in}^{\mathcal{M}}$, $\text{all-equals-in}^{\mathcal{M}}$ take two formulas in $\boldsymbol{A}$ as argument and model the relations *implies*, *related*, *all-in-part*, *all-in-whole*, *some-implies-in*, *all-implies-in*, *all-equals-in* between these two formulas in FOL, respectively.
- The predicates $=^{\mathcal{M}}$ and $<^{\mathcal{M}}$ take two real numbers as arguments and model the relations *equal to* and *strictly less than*, respectively.
- The predicate $\text{eval}^{\mathcal{M}}$ takes one formula *fm* in $\boldsymbol{A}$ as argument and holds for *fm* iff *fm* is evaluated as true in the model under consideration in FOL.
- Let xs and ys be formulas in CNF from $\boldsymbol{A}$, and f is a function $f : \text{set}(xs) \to \text{set}(ys)$

$$\text{implies}(xs, ys) =_{\text{def}} \text{eval}(xs) \to \text{eval}(ys) \tag{25}$$

$$\text{related}(x, y) =_{\text{def}} \text{implies}(x, y) \vee \text{equals}(x, y) \quad \text{(see footnote}^1\text{)} \tag{26}$$

$$\text{all-in-part}(xs, ys) =_{\text{def}} (\forall x \in \text{set}(xs) : \text{related}(x, f(x))) \wedge (|xs| < |ys|),\ f \text{ is injective} \tag{27}$$

$$\text{all-in-whole}(xs, ys) =_{\text{def}} (\forall x \in \text{set}(xs) : \text{related}(x, f(x))) \wedge (|xs| = |ys|),\ f \text{ is bijective} \tag{28}$$

$$\text{some-implies-in}(xs, ys) =_{\text{def}} \exists x \in \text{set}(xs) : \text{implies}(x, f(x)) \wedge \neg\text{equals}(x, f(x)) \tag{29}$$

$$\text{all-implies-in}(xs, ys) =_{\text{def}} \forall x \in \text{set}(xs) : \text{implies}(x, f(x)) \wedge \neg\text{equals}(x, f(x)) \tag{30}$$

$$\text{all-equals-in}(xs, ys) =_{\text{def}} \forall x \in \text{set}(xs) : \text{equals}(x, f(x)) \tag{31}$$

[1] If two formulas have the same predicate symbols and arguments, these two formulas are equal.

Then, we have the following mappings:

- $\text{all-in-whole}(P_1, P_2) \wedge \text{some-implies-in}(P_1, P_2)$ iff R_1 refines R_2
- $\text{all-in-part}(P_1, P_2) \wedge \text{all-implies-in}(P_1, P_2)$ iff R_1 partially-refines R_2
- $\text{all-in-part}(P_2, P_1) \wedge \text{all-equals-in}(P_2, P_1)$ iff R_1 contains R_2

We have the following properties for these relations between formulas:

- *all-in-whole*, *all-in-part*, *all-implies-in*, and *some-implies-in* relations are transitive
- *all-in-part* and *all-in-whole* relations are disjoint
- *all-equals-in* and *some-implies-in* are disjoint
- *all-equals-in* and *all-implies-in* are disjoint

We have the following inferences for the relations between formulas:

- $(P_1 \text{ all-in-part } P_2) \wedge (P_2 \text{ all-in-whole } P_3) \to (P_1 \text{ all-in-part } P_3)$
- $(P_1 \text{ all-in-whole } P_2) \wedge (P_2 \text{ all-in-part } P_3) \to (P_1 \text{ all-in-part } P_3)$
- $(P_1 \text{ some-implies-in } P_2) \wedge (P_2 \text{ all-implies-in } P_3) \to (P_1 \text{ all-implies-in } P_3)$
- $(P_1 \text{ all-implies-in } P_2) \wedge (P_2 \text{ some-implies-in } P_3) \to (P_1 \text{ all-implies-in } P_3)$
- $(P_1 \text{ some-implies-in } P_2) \wedge (P_2 \text{ all-equals-in } P_3) \to (P_1 \text{ some-implies-in } P_3)$
- $(P_1 \text{ all-implies-in } P_2) \wedge (P_2 \text{ all-equals-in } P_3) \to (P_1 \text{ all-implies-in } P_3)$
- $(P_1 \text{ all-equals-in } P_2) \wedge (P_2 \text{ all-implies-in } P_3) \to (P_1 \text{ all-implies-in } P_3)$

Based on the mappings, we note that we always have *some-implies-in*, *all-implies-in*, and *all-equals-in* relations with *all-in-part* and *all-in-whole* relations in the mapping. For instance, from the mapping, it can be seen that the *some-implies-in* relation occurs with *all-in-whole*. With inferences of these relations, it can be inferred that the *some-implies-in* relation occurs with *all-in-part* but *some-implies-in* and *all-in-part* relations together are not mapped to any requirements relation. Therefore, the *some-implies-in* relation always occurs with *all-in-part* or *all-in-whole* relations.

Appendix C: Part of the CMS requirements document

In this appendix, we give an overview of the requirements of the CMS as used in this paper. The full requirements document is available at http://wwwhome.cs.utwente.nl/~goknila/sosym/.

Requirements (partial)

Stakeholder general

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 facility

Stakeholder students

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

R11: The system shall enable students to subscribe to and unsubscribe from courses

R16: The system shall allow messages to be sent to individuals, teams, or all course participants at once

R24: The system shall notify students about events (new messages posted, etc.)

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

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

Stakeholder lecturers

R48: The system shall allow lecturers to create courses

R49: The system shall allow lecturers to create entirely new courses

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

Stakeholder administration

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

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

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

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

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

Glossary (partial)

Static course information: Information about a course which does not change while a course is given but does change between semesters. This includes the lecturer, number of ECTS credits, and study material

Dynamic course information: Information about a course which changes while a course is given. This includes news messages, archived files, and roster

Manage courses: Managing courses involves the creation, reading, updating, and deleting of courses

References

1. Aizenbud-Reshef, N., Paige, R.F., Rubin, J., Shaham-Gafni, Y., Kolovos, D.S.: Operational semantics for traceability. In: ECMDA-TW 2005, Nuremberg, pp. 7–14 (2005)
2. Baudry, B., Nebut, C., Le Traon, Y.: Model-driven engineering for requirements analysis. In: EDOC 2007, pp. 459–466. IEEE Computer Society Press, Annapolis (2007)
3. Brottier, E., Baudry, B., Le Traon, Y., Touzet, D., Nicolas, B.: Producing a global requirement model from multiple requirement specifications. In: EDOC 2007, pp. 390–404. IEEE Computer Society Press, Annapolis (2007)
4. Borland Caliber Analyst. http://www.borland.com/us/products/caliber/index.html
5. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., Natt och Dag, J.: An industrial survey of requirements interdependencies in software product release planning. In: Proceedings of the 5th International Symposium on Requirements Engineering, Toronto, Canada, pp. 84–91 (2001)
6. Ceron, R., Duenas, J.C., Serrano, E., Capilla, R.: A meta-model for requirements engineering in system family context for software process improvement using CMMI. In: PROFES 2005. Lecture Notes in Computer Science, vol. 3547, pp. 173–188. Springer, Berlin (2005)
7. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Reading (2000)
8. Dahlstedt, A.G., Persson, A.: Requirements interdependencies: state of the art and future challenges. In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements, pp. 95–116. Springer, Berlin (2005)
9. Dean, M., Schreiber, G., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A.: OWL Web Ontology Language Reference W3C Recommendation (2004)
10. Duffy, D., MacNish, C., McDermid, J., Morris, P.: A framework for requirements analysis using automated reasoning. In: Livari, J., Rossi, M., Lyytinen, K. (eds.) CAiSE 1995. Lecture Notes in Computer Science, vol. 932, pp. 68–81. Springer, Berlin (1995)
11. Eclipse Rich Client Platform (RCP). http://www.eclipse.org/home/categories/rcp.php
12. Egyed, A., Grunbacher, P.: Supporting software understanding with automated requirements traceability. Int. J. Softw. Eng. Knowl. Eng. **15**(5), 783–810 (2005)
13. Egyed, A., Grunbacher, P.: Automated requirements traceability: beyond the record and replay paradigm. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE), Edinburgh, Scotland, pp. 163–171 (2002)
14. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. **20**(8), 569–578 (1994)

15. Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Formal reasoning techniques for goal models. In: Journal on Data Semantics, Lecture Notes in Computer Science, vol. 2800, pp. 1–20. Springer, Berlin (2003)
16. Goknil, A., Kurtev, I., van den Berg, K.: A metamodeling approach for reasoning about requirements. In: European Conference on Model Driven Architecture (ECMDA 2008), Berlin, Germany. Lecture Notes in Computer Science, vol. 5095, pp. 310–325. Springer, Berlin (2008)
17. Goknil, A., Kurtev, I., van den Berg, K.: Change impact analysis based on formalization of trace relations for requirements. In: 4th ECMDA Traceability Workshop (ECMDA-TW 2008), Berlin, Germany, pp. 59–75 (2008)
18. Goknil, A.: Tutorial: requirements relations and definitions with examples. http://www.home.cs.utwente.nl/~goknila/tutorial/Relations_Tutorial.doc
19. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Proceedings of the First International Conference on Requirements Engineering, pp. 94–101. IEEE Computer Society Press, Colorado (1994)
20. Guide to Software Engineering Body of Knowledge. IEEE Computer Society, Colorado. http://www.swebok.org/. Accessed 19 October 2009
21. Heaven, W., Finkelstein, A.: UML profile to support requirements engineering with KAOS. IEE Proc. Softw. **151**(1), 10–27 (2004)
22. Heitmeyer, C.L., Jeffords, R.D., Labaw, G.L: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (1996)
23. Hunter, A., Nuseibeh, B.: Managing inconsistent specifications: reasoning, analysis, and action. ACM Trans. Softw. Eng. Methodol. **7**(4), 335–367 (1998)
24. Huth, M.R.A., Ryan, M.D: Logic in Computer Science: Modeling and Reasoning about Systems. Cambridge University Press, Cambridge (2000)
25. IBM Rational RequisitePro. http://www-01.ibm.com/software/awdtools/reqpro/
26. IBM Telelogic Doors. http://www.telelogic.com/Products/doors/doors/index.cfm
27. INCOSE Requirements Management Tool Survey. http://www.incose.org
28. JGraph—Java Graph Visualization and Layout. http://www.jgraph.com/
29. Jena—A Semantic Web Framework for JAVA. http://jena.sourceforge.net/
30. Koch, N., Kraus, A.: Towards a common metamodel for the development of web applications. In: ICWE 2003. Lecture Notes in Computer Science, vol. 2722. pp. 497–506. Springer, Heidelberg (2003)
31. Lopez, O., Laguna, M.A., Garcia, F.J.: Metamodeling for requirements reuse. Anais do WER02—Workshop em Engenharia de Requisitos. Valencia, Spain, pp. 76–90 (2002)
32. Meyer, J.J.C., Wieringa, R., Dignum, F.: The Role of Deontic Logic in the Specification of Information Systems. Logics for Databases and Information Systems, pp. 71–115 (1998)
33. Moon, M., Yeom, K., Chae, H.S.: An approach to developing domain requirements reuse as a core asset based on commonality and variability analysis in a product line. IEEE Trans. Softw. Eng. **31**(7), 551–569 (2005)
34. Mylopoulos, J., Chung, L., Yu, E.: From object-oriented to goal oriented requirements analysis. Commun. ACM **42**(1), 31–37 (1999)
35. Navarro, E., Mocholi, J.A., Letelier, P., Ramos, I.: A metamodeling approach for requirements specification. J. Comput. Inf. Syst. **46**(5), 67–77 (2006)
36. Noppen, J., van den Broek, P., Aksit, M.: Imperfect requirements in software development. In: REFSQ 2007. Lecture Notes in Computer Science, vol. 4542. pp. 247–261. Springer, Heidelberg (2007)
37. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE Trans. Softw. Eng. **20**(10), 760–773 (1994)
38. OMG: SysML Specification. OMG ptc/06-05-04, http://www.sysml.org/specs.htm
39. Ramesh, B., Jarke, M.: Toward reference models for requirements traceability. IEEE Trans. Softw. Eng. **27**(1), 58–93 (2001)
40. Rashid, A., Moreira, A., Araujo, J.: Modularization and composition of aspectual requirements. In: AOSD 2003, Boston, USA, pp. 11–20 (2003)
41. Requirements Interchange Format (RIF), http://www.automative-his.de/rif/doku.php
42. Robinson, W.N., Pawlowski, S.D., Volkov, V.: Requirements interaction management. ACM Comput. Surv. **35**(2), 132–190 (2003)
43. Rodrigues, O., Garcez, A., Russo, A.: Reasoning about requirements evolution using clustered belief revision. In: Bazzan, A.L.C., Labidi, S. (eds.) SBIA 2004. Lecture Notes in Computer Science (LNAI), vol. 3171, pp. 41–51. Springer, Berlin (2004)
44. Soares, M.S., Vrancken, J.: Model-driven user requirements specification using SysML. J. Softw. **3**(6), 57–68 (2008)
45. Supakkul, S., Chung, L.: A UML profile for goal-oriented and use case driven representation of NFRs and FRs. In: SERA 2005, pp. 112–119. IEEE Computer Society Press, Michigan (2005)
46. ten Hove, D., Goknil, A., Kurtev, I., van den Berg, K., de Goede, K.: Change impact analysis for SysML requirements models based on semantics of trace relations. In: Fifth ECMDA Traceability Workshop (ECMDA-TW 2009), Enschede, the Netherlands, pp. 17–28 (2009)
47. Tool for Requirements Inferencing and Consistency Checking (TRIC). http://trese.cs.utwente.nl/tric/
48. TopTeam Analyst. http://www.technosolutions.com/topteam_requirements_management.html
49. van Lamswerdee, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. IEEE Trans. Softw. Eng. **24**(11), 908–926 (1998)
50. van Lamswerdee, A.: Goal-oriented requirements engineering: a roundtrip from research to Practice. In: Invited Minitutorial, Proceedings RE'01—5th International Symposium Requirements Engineering, Toronto, pp. 249–263 (2001)
51. Veldhuis, J.W.: Tool support for a metamodeling approach for reasoning about requirements. MSc Thesis, University of Twente. http://essay.utwente.nl/58693/1/scriptie_J_Veldhuis.pdf (2009). Toolversion 1.0 available from http://trese.cs.utwente.nl/tric/
52. Vicente-Chicote, C., Moros, B., Toval, A.: REMM-Studio: an integrated model-driven environment for requirements specification, validation and formatting. J. Object Technol. (Special Issue TOOLS Europe) **6**(9), 437–454 (2007)
53. von Knethen, A., Paech, B.: A survey on tracing approaches in practice and research. IESE-Report, No. 095.01/E, version 1.0, Fraunhofer Institut Experimentelles Software Engineering (2002)
54. Zowghi, D., Offen, R.: A logical framework for modeling and reasoning about the evolution of requirements. In: RE 1997, Annapolis, USA, pp. 247–257, January (1997)
55. Zowghi, D., Gervasi, V.: On the interplay between consistency, completeness and correctness in requirements evolution. Inf. Softw. Technol. **45**, 993–1009 (2003)

Author Biographies

Arda Goknil is a PhD candidate at the Computer Science Department of the University of Twente in The Netherlands. Goknil received his BSc degree in 2003, MSc degree in 2006, from the Computer Engineering Department of Ege University in Turkey. His research interests include model-driven engineering, traceability, requirements and architecture modeling, and change impact analysis. He carries out his PhD research in the context of the QuadREAD-project (Quality-Driven Requirements Engineering and Architectural Design).

Ivan Kurtev holds a MSc degree in Computer Science from University of Sofia and a PhD degree in Software Engineering from University of Twente. He is currently an assistant professor in the Software Engineering group in University of Twente, the Netherlands. His main research interests are in the domain of Model Driven Engineering with a focus on model transformation languages, metamodeling, requirements modeling and traceability.

Klaas van den Berg is an Assistant Professor in the Software Engineering Group of the University of Twente (the Netherlands). He received his BSc and MSc in Electrical Engineering and his PhD degree in Computer Science from the University of Twente. He has been lecturer for some years at the University of Zambia and the University of Dar es Salaam. Since 1986, he teaches graduate and undergraduate courses on Software Engineering and Software Management in the Department of Computer Science at the University of Twente. His research interests include software measurement and quality metrics, empirical software engineering, software process modeling, software evolvability, traceability and model-driven engineering. As responsible and senior researcher, he is involved in several research projects related to these subjects. He participated in numerous workshops and conferences as organizer and member of the program committee.

Jan-Willem Veldhuis received his BSc and MSc degrees in Computer Science from University of Twente, the Netherlands. He developed TRIC (Tool for Requirements Inferencing and Consistency Checking) in his master project. He works as an application developer at Oracle in the Netherlands.