Software Quality Journal (c) Springer 2013

KBRE: A Framework for knowledge-based Requirements Engineering

Tuong Huan Nguyen · Bao Quoc Vo · Markus Lumpe · John Grundy

Received: 21 September 2012 / Accepted: 26 March 2013

Abstract Detecting inconsistencies is a critical part of Requirements Engineering (RE) and has been a topic of interest for several decades. Domain knowledge and semantics of requirements not only play important roles in elaborating requirements but are also a crucial way to detect conflicts among them. In this paper we present a novel knowledge-based RE framework (**KBRE**) in which domain knowledge and semantics of requirements are central to elaboration, structuring, and management of captured requirements. Moreover, we also show how they facilitate the identification of requirements inconsistencies and other related problems. In our KBRE model, Description Logic (DL) is used as the fundamental logical system for requirements analysis and reasoning. In addition, the application of DL in the form of Manchester OWL Syntax brings simplicity to the formalization of requirements while preserving sufficient expressive power. A tool has been developed and applied to an industrial use case to validate our approach.

Keywords Requirements Engineering, Inconsistencies, Identification, Description Logics, Manchester OWL Syntax, Ontology

1 Introduction

Requirements Engineering (RE) is an iterative process of eliciting, elaborating, structuring, specifying, analyzing, and managing requirements of stakeholders on a software system (Sommerville, 2011; Pressman, 2005; Van Lamsweerde, 2001). Inconsistencies or conflicts between requirements have been known to present a key challenge to requirements engineers during the RE process. These can arise during or be-

Tuong Huan Nguyen · Bao Quoc Vo · Markus Lumpe · John Grundy
Faculty of Information and Communication Technology
Swinburne University of Technology
Melbourne - Australia
E-mail: huannguyen@swin.edu.au, bvo@swin.edu.au, mlumpe@swin.edu.au, jgrundy@swin.edu.au

tween different phases of software development (Boehm and In, 1999). Two or more requirements are inconsistent if they cannot be satisfied simultaneously by the system being developed without sacrificing at least one of them. Inconsistencies are considered a serious problem in RE because it requires removing or modifying requirements to ensure that a conflict-free requirement set is produced at the end of the software development process. Moreover, this can be a potential cause of project failure since removing or modifying a requirement can directly affect the correctness of requirements and thus, the success of the project. A study of a number of large projects in 2006 (Henderson, 2006) has pointed out that requirements related issues, including inconsistencies, are among the major reasons causing project failures. In addition, research reports from the Standish Group (Doe, 2009) also emphasize requirements inconsistencies and stakeholders' conflicts in specifying system requirements as a serious issue in RE.

There have been numerous techniques proposed in response to the industry's need for improved RE processes over the last two decades (Boehm et al., 1995; Robinson and Pawlowski, 1999; Egyed and Grunbacher, 2004; Dardenne et al., 1993; Van Lamsweerde, 2001; Fuxman et al., 2004; Chung, 1993; Lauenroth and Pohl, 2008; Grundy et al., 1998; Kamalrudin et al., 2010; Weston et al., 2009). We are particularly interested in logic-based approaches to identify conflicts, as they entail well-defined inconsistency detection procedures. This is due to the formalization of requirements using logical languages and also because it enables checking arbitrary inconsistency rules (Spanoudakis and Zisman, 2001). However, most existing research within this paradigm either lacks or insufficiently supports the creation and maintenance of domain knowledge and semantics of requirements. These play important roles in specifying system requirements (Zave and Jackson, 1997; Kenzi et al., 2010; Kaiya et al., 2010). Requirements can be viewed as *optative* statements that capture stakeholder demands, whose understanding requires *domain knowledge* to help bridge between a stakeholder's design on what a system needs to do and what is practically implementable in that system. The domain knowledge, which also contains rules and assumptions about the system's operating environment, offers us a practical means to identify inconsistencies and overlaps in requirements that may arise from the competing objectives and/or different stakeholders' preferences. Some types of requirement inconsistencies may not be detectable in the absence of such domain knowledge (Boehm et al., 1995; Robinson and Pawlowski, 1999). Consider, for example, a scenario stipulating conflicting quality attributes for bitmaps in a graphical user interface. The requirements "*the application must support bitmaps of at least 1280x960 pixels*" and "*the size of an individual bitmap must not exceed 3MB*" cannot be satisfied simultaneously in the presence of the domain knowledge that also includes the rule "*bitmaps can require a color-depth of up to 24 bits per pixel*."

Moreover, reasoning about requirements typically demands semantics of the requirements and the related concepts and constraints be defined using instances, concepts and roles (Breaux et al., 2008). In knowledge representation, concepts and roles are defined abstractly in the domain-level knowledge base using sub-class or specialization relationships. For instance, when specifying requirements, there may be several terms referring to the same concept, such as *customer* and *client*. In order to determine if inconsistencies exist and to aid requirements elicitation, it is necessary that these concepts are formally defined. For example, by using an ontology that gives precise meanings to the terms used to specify requirements. Furthermore, the relationships between concepts and instances also need to be taken into account when performing analysis on the captured requirements. Consider two requirements regarding the languages supported by an enterprise-wide system: one states that "*Chinese must be supported*" and another says that "*only major languages should be supported*." The interaction (*e.g.*, conflict) between the two requirements cannot be determined precisely without knowing about the relationship between *Chinese* and the concept of *major languages* (*i.e.*, whether Chinese is one of the major languages).

Finally, many existing approaches provide little or no explanations of the detected requirements conflicts. From our point of view, good explanation given to requirements engineers and stakeholders about detected requirements conflicts are critical. This is because they give requirements engineers and stakeholders a clear idea of where the conflicts come from, why they are conflicting, and help point to possible resolutions to the problems.

In this paper, we propose a new goal-directed RE process that addresses the above mentioned problems and describe our knowledge-based RE framework (**KBRE**). In the KBRE model, domain knowledge and semantics of requirements are centralized. Using these, requirements can be elicited, elaborated, and inconsistencies and other related requirements problems can be detected. We use the description logic *SROIQ* (Horrocks et al., 2006) as the fundamental logical system for analysis of and reasoning about requirements. In particular, we focus on identifying inconsistencies between requirements as well as redundancies and overlaps of requirements. In addition, the use of description logic codified in the form of Manchester OWL syntax (MOS) (Horridge et al., 2006),[1] as the requirements specification language, facilitates the creation and maintenance of the ontologies for storing the domain knowledge and semantics of requirements. Furthermore, the application of MOS brings more simplicity to our framework while preserving sufficient expressive power. More precisely, MOS offers us a means to reduce the potential risk of creating *pragmatic barriers* (Dwyer et al., 1999). In addition, MOS allows the successful application of our approach as it seeks for balance the efficiency of a formal language with the expressiveness of a natural language (Guarino et al., 2009). Using description logic also allows us to utilize an off-the-shelf OWL reasoner Pellet (Clark & Parsia, LCC, 2012) with some extensions, as a tool for complete and sound conflict detection and to provide explanation services. Moreover, the KBRE model was developed based on the Goal-oriented Requirements Engineering approach (Anton, 1996; Van Lamsweerde, 2001) to improve the manageability of requirements and to facilitate the traceability of the underlying rationale of inconsistent requirements. To demonstrate the feasibility of our RE model, we have also developed *REInDetector*, a prototype knowledge-based requirements engineering tool that supports automatic detection of a range of inconsistencies. The tool and a user guide are available for download and experimentation with from: `http://www.ict.swin.edu.au/personal/huannguyen/REInDetector.html`

[1] In our model, expressions in Manchester Syntax can be translated into Description Logics automatically and vice versa.

The rest of the paper is structured as follows. Section 2 presents some background knowledge of this work. Section 3 provides a discussion of the new goal-directed RE process that we propose. The analysis and reasoning services provided by this model are presented in Section 4. In Section 5, we discuss the *REInDetector*tool that we have developed to realize and demonstrate our proposed framework. The evaluation of the tool with a simple use case can be found in 6. Section 7 contains the reviews on related works in the problem of requirements conflict identification. In Section 8, we conclude our contributions and point out the future works. Details about the *REInDetector*tool can be found in the Appendix.

2 Background

2.1 Goal Oriented Requirements Engineering

Most traditional Requirements Engineering (RE) approaches have been shown to be inadequate in dealing with complex software systems engineering(DeMarco, 1979; Ross, 1977; Rumbaugh et al., 1991). These approaches treat requirements as consisting of only processes and data and do not capture the rationale behind them, thus making it difficult to understand requirements with respect to some high-level concerns in the problem domain. In addition, most techniques focus on modeling and specification of the software alone and therefore, lack support for reasoning about the overall system comprising both the system-to-be and its environment (Rumbaugh et al., 1991).

The Goal-oriented Requirements Engineering (GORE) approach was developed in an attempt to overcome these problems. GORE is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements (Van Lamsweerde, 2001). The idea of GORE is derived from the recognition that goals are the root from which all requirements of a project can be defined. In fact, the system under consideration is typically analyzed in its organizational, operational and technical settings. As soon as problems are pointed out and opportunities are identified, high-level goals can be defined and possibly refined into sub-goals to address those problems and to meet the identified opportunities. Subsequently, requirements are elaborated to satisfy these goals. Requirements can also be further refined into more specific requirements to meet the higher-level requirements. Other requirements are then elicited by repeating this process. GORE addresses the key issues associated with a lack of requirements' rationale by providing both top-down and bottom-up refinement and operationalizations between requirements. It is thus possible to point out the underlying reason for this requirement (through links to higher-level requirements) and how it is realized by the system being developed (through links to lower-level requirements). The main benefits of a GORE-based approach include:

- Goals provide a precise criterion for sufficient completeness of a requirements specification. That is, the specification is complete with respect to a set of goals if all the goals can be proved to be achieved from the specification and the properties known about the domain considered (Yu, 1993).

- Goals help to avoid capturing irrelevant requirements. That is, goals provide a precise criterion for requirements pertinence: a requirement is pertinent with respect to a set of goals in the domain considered if its specification is used in the proof of at least one goal (Yu, 1993).
- GORE also helps when explaining requirements to stakeholders. Goals provide the rationale for requirements, in a way similar to design goals in design processes (Lee, 1991; Mostow, 1985). A requirement appears because of some underlying goal which provides a base for it (Ross, 1977; Sommerville and Sawyer, 1997; Dardenne et al., 1991).
- Goal refinement provides a natural mechanism for structuring complex requirements for increased readability.
- Requirements engineers are faced with many options to consider during the requirements elaboration process. Van Lamsweerde (Van Lamsweerde, 2001) has pointed out that alternative goal refinements provide the right levels of abstraction at which decision makers can be involved to validate choices being made or suggest other overlooked alternatives. Alternative goal refinements also allow alternative proposals for the realization of a system to be explored (Van Lamsweerde, 2000).

2.2 Description Logics

Description Logics (DLs) (Baader et al., 2007) are a family of knowledge representation languages which are decidable subsets of first-order logic. DLs are commonly used as a formal basis for object/class-style ontology languages. We propose to use description logic as the underlying framework for formalizing, analyzing and reasoning about requirements in the GORE process for the following key reasons:

- Although other formal languages such as temporal logics or first-order logic can also be used to specify requirements and to provide a reasoning framework for supporting the analysis of requirements, they are too complex syntactically to be accessible for requirements engineers in practice or computationally undecidable, or both.
- With the objectives of creating and maintaining domain knowledge and semantics for requirements that require the concepts and relationships in the problem domain to be defined, DLs provide a rich formal semantics for ontology specification languages such as various members of the Web Ontology Languages (OWL) (Corcho and Gómez-Pérez, 2000).
- Description logic fits nicely to the need of representing the relations between concepts, roles and instances in GORE in an intuitive manner.

In this section, we provide an overview of the $\mathcal{SROIQ}$ description logic (Horrocks et al., 2006). We choose to use $\mathcal{SROIQ}$ logic because of its expressiveness. This allows declarations of disjoint roles[2] (i.e. if two classes are related by a binary role then they must not be related by another binary role which is disjoint with the previous role), reflexive and irreflexive roles and negated role assertions. These forms of

[2] In description logics, roles represent binary relationships between classes.

expressions are critical in order to build relationships between concepts and roles in a problem domain and are not fully supported in other description logics. Furthermore, the *SROIQ* logic corresponds to OWL-2, the newest version in the family of OWL languages.

In *SROIQ* logic, the domain of interest is modeled by individuals, concepts and roles. Individuals are the instances that instantiate particular concepts, thus concepts can be viewed as representing unary properties of individuals, while roles consist of binary relations between concepts or individuals. In this section, we use C and D to denote concept expressions. A concept inclusion axiom is an expression of the form $C \sqsubseteq D$ (which means the concept C is a specialization of the concept of D). For instance, the class (or, concept in description logics terminologies) AcademicStaff is inclusive in the class Staff, denoted by AcademicStaff $\sqsubseteq$ Staff. A *SROIQ* TBox is a set of general concept inclusion axioms. Thus, a TBox can capture a class hierarchy. On the other hand, an ABox is a finite set of concept expressions of the form $C(a)$ (which means a is an instance of C). For instance, to express that James is an academic staff member, we write AcademicStaff(James). Or, to express that Mary is a friend of James, we write friendOf(James, Mary). Finally, all assertions concerning roles are gathered in a RBox (which is introduced only in *SROIQ*). For instance, to indicate that the relationship hasPart translates the ownership of an object (e.g., a car) to the ownership of its parts (e.g., the car's engine), we can state the following role inclusion axiom in an RBox: own $\circ$ hasPart $\sqsubseteq$ own. The *SROIQ* knowledge base is the union of the TBox, ABox and RBox. An interpretation I consists of a set Δ^I called the interpretation domain and an interpretation function $.^I$. The interpretation function assigns to each atomic concept A a subset of Δ^I, each role R a subset of $\Delta^I \times \Delta^I$ and each individual a an element of Δ^I. Table 1 illustrates the syntax of *SROIQ* logic.

Name	Syntax	Semantics
inverse role	R^-	$\{\langle x,y\rangle \in \Delta^I \times \Delta^I \mid \langle y,x\rangle \in R^I\}$
universal role	U	$\Delta^I \times \Delta^I$
top	$\top$	Δ^I
bottom	$\perp$	$\emptyset$
negation	$\neg C$	$\Delta^I \backslash C^I$
conjunction	$C \sqcap D$	$C^I \cap D^I$
disjunction	$C \sqcup D$	$C^I \cup D^I$
nominals	$\{\alpha\}$	$\{\alpha^I\}$
univ.restriction	$\forall R.C$	$\{x \in \Delta^I \mid \langle x,y\rangle \in R^I$ implies $y \in C^I\}$
exist.restriction	$\exists R.C$	$\{x \in \Delta^I \mid$ for some $y \in \Delta^I$, $\langle x,y\rangle \in R^I$ and $y \in C^I\}$
Self concept	$\exists S.Self$	$\{x \in \Delta^I \mid \langle x,x\rangle \in S^I\}$
qualified number	$\leq nS.C$	$\{x \in \Delta^I \mid \#\{y \in \Delta^I \mid \langle x,y\rangle \in S^I$ and $y \in C^I\} \leq n\}$
restriction	$\geq nS.C$	$\{x \in \Delta^I \mid \#\{y \in \Delta^I \mid \langle x,y\rangle \in S^I$ and $y \in C^I\} \geq n\}$

Table 1 Semantics of concept constructors in *SROIQ* for an interpretation I with domain Δ^I (Horrocks et al., 2006)

In general, *SROIQ* logic is mainly used to represent concepts, instances and their relationships. Table 2 presents the examples adapted from (Horrocks et al., 2006) to illustrate the use of *SROIQ* logic as a formal representation language.

SROIQ constructors	Examples	Meaning
Conjunction ($\sqcap$)	UserWithLockedAccount $\equiv$ User $\sqcap$ $\exists$hasAccount.LockedAccount	A user-with-locked-account is a user who has had his account locked
Disjunction ($\sqcup$)	Messaging $\equiv$ InstantMessaging $\sqcup$ AsynchronousMessaging	A type of messaging can be either instant or asynchronous
Negation ($\neg$)	User $\equiv$ $\neg$ Admin	A user account is not an admin account of the system
Inclusion ($\sqsubseteq$)	ProfilePicture $\sqsubseteq$ Photo	A profile picture is a photo
Nominal ($\{\}$)	$\{$English$\}$	An individual named 'English' (which is a language in this domain)
Universal restriction ($\forall$)	User $\sqcap$ $\forall$hasPhoto.LandscapePhoto	The class of users whose photos are all landscape photos
Existence restriction ($\exists$)	SocialSystem $\sqsubseteq$ $\exists$ supportLanguage.MainLanguage	A social networking system needs to support at least one main language

Table 2 Examples of *SROIQ* logic

3 A process model for Requirements Engineering - KBRE

In this section, we describe our new RE process model, the *Knowledge-Based Requirements Engineering* methodology (or, **KBRE**), that supports elaboration, management and analyses of requirements goals. We first provide an overview of the model and then present the language used for specifying requirements. We also describe the goal refinement constructs and the syntax for specifying goals and their relationships.

3.1 Model overview

The KBRE model is adapted from the one used in KAOS (Van Lamsweerde et al., 1998). In KBRE, the focus is on the refinement, formalization and analysis of goals. Therefore, compared to KAOS, it does not support the definition of agents and responsibility assignments.

The KBRE method is derived from the Goal oriented Requirements Engineering approach. It supports the capturing of the system's functional and non-functional requirements and shows how they are related to each other through refinement and operationalization links. In addition, it enables the detection of a number of problems in captured requirements, including inconsistencies between requirements and redundancies and overlaps of requirements.

In KBRE, goals are the high-level expectations of stakeholders in the system being developed and requirements are referred to the lowest level of expectations of stakeholders which are refined/operationalized in a goal tree. Currently, KBRE only supports analysis and reasoning with low-level goals for the following two reasons. First, low-level goals contain sufficient details and information at the domain level to enable analysis that is meaningful to requirements engineers. Second, high-level goals are normally vague. The identification of inconsistencies related to these goals requires the understanding of requirements engineers in particular context and thus is beyond the current support of our ontology. For instance, *'Building a cost-effective system'* is a vague goal. Assume that there is a requirement that *'Support video conferences and instant messaging*. If implementing video conferences and instant messaging is very costly, it could prevent the *'cost-effective* goal being achieved. Such a situation needs to be identified as a potential inconsistency. However, in order to identify such an inconsistency, the ontology needs to somehow represent the *'contradictory* relationship between *'supporting video conferences and instant messaging* and *'having cost-effective system'*. We are currently working on an extension of KBRE to enable it to include a meta-model for the this type of relationship to enable reasoning at higher levels of the requirements hierarchy. This extension, however, is still part of our work in progress and will not be discussed in this paper.

Graphically, a goal model is represented by an AND/OR graph, called a *goal graph*. In the goal graph, each individual goal is represented as a node that is annotated according to the goal's feature. Each goal node is connected to other goals in the same graph via edges. A set of related edges (i.e. edges connecting a parent-goal to its sub-goals) represents a refinement or an operationalization. A refinement not only indicates how a goal is decomposed into sub-goals, i.e. how a goal is realized, but also reveals the parent goal it comes from, showing the rationale behind the goal.

3.2 A Requirements Specification Language

Closely based on Manchester OWL Syntax (MOS) (Horridge and Patel-Schneider, 2009), our requirement specification language has the following advantages. First, it is a user-friendly specification language since the formal expressions are constructed from English words and therefore can be easily understood and communicated by humans. Second, MOS expressions can be automatically translated into description logics (and vice versa) and allow requirements to be formally analyzed and reasoned about. Lastly and most importantly, by using Manchester OWL Syntax, we can add semantics to requirements and store domain knowledge by building and maintaining an ontology of concepts and roles for the problem domain.

The semantics and syntax of our proposed specification language are the same as those of MOS except that it borrows the OWL language constructor 'SubClassOf', which is named *specifier* in this language, to achieve more expressive power for representing requirements. In addition, the symbol "%" is introduced to allow concepts to be defined within a requirement's formalization.

A requirement is usually represented by one or more sentences. Each sentence consists of two expressions connected by the specifier. The expression on the right-hand side generally denotes the expectation, or constraints on the concepts or individuals represented in the expression on the left-hand side of the specifier. The following example illustrates the use of this language.

Example 1. The requirement *'The system supports Photo Upload'* is formalized as: `'System SubClassOf supportFeature VALUE PhotoUpload.'`

The requirement *'Photos must not exceed 4Mb in size'* is formalized as: `'Photo SubClassOf hasSizeInMb MAX 4.'`

Apart from requirements, MOS is also used to represent knowledge in the domain in a similar way. Example 2 shows how the rule *bitmaps can require a color-depth of up to 24 bits per pixel* is represented

Example 2. `'Bitmap SubClassOf hasDepthInBitsperpixel MAX 24.'`

3.3 Ontology

In KBRE, to facilitate analysis of and reasoning about requirements, we augment the specifications of requirements with semantics and domain knowledge in the form of an ontology.

3.3.1 Ontology Definition and Representation

An ontology can be considered as a repository of concepts, roles and instances in the requirements' domain. In the OWL language, an ontology is referred to as a collection of the definitions of a number of classes, properties, individuals and their relationships. As discussed earlier, in KBRE Manchester OWL Syntax is used for specifying requirements. In this language, each requirement specification is made up from classes, individuals and properties. An ontology will be created and kept updated during the RE process to store the definitions and relationships of those classes, properties and individuals to enable the analysis and reasoning on the requirements. From this section to the end of the paper, we use the term ontology entities to refer to concepts, properties, individuals and their relationships in an ontology.

Figure 1 shows the main parts of the meta-model of our ontology. As can be seen from the figure, *class*, *individual* and *property* are three main ontology entities. Classes are categorized according to the way they are constructed. A class can be constructed from boolean combinations of other classes with *Complement*, *Intersection* and *Union*. A class also can be created either from the restrictions that apply constraints on the range of a property for the context of a class or direct enumerations of named individuals. There are two different types of properties, namely object properties and data properties. A property always has its domain as a class. An object property has its range as a class while a data property's range is a data type. An individual is an instantiation of a class and is the subject of a *Property Value*, which instantiates a property. An individual can be an object of an *Object Property Value* while the subject of a *Data Property Value* must be a *Data Value*. Readers who are interested in meta-model of OWL ontology can find more details in (Brockmans et al., 2004).

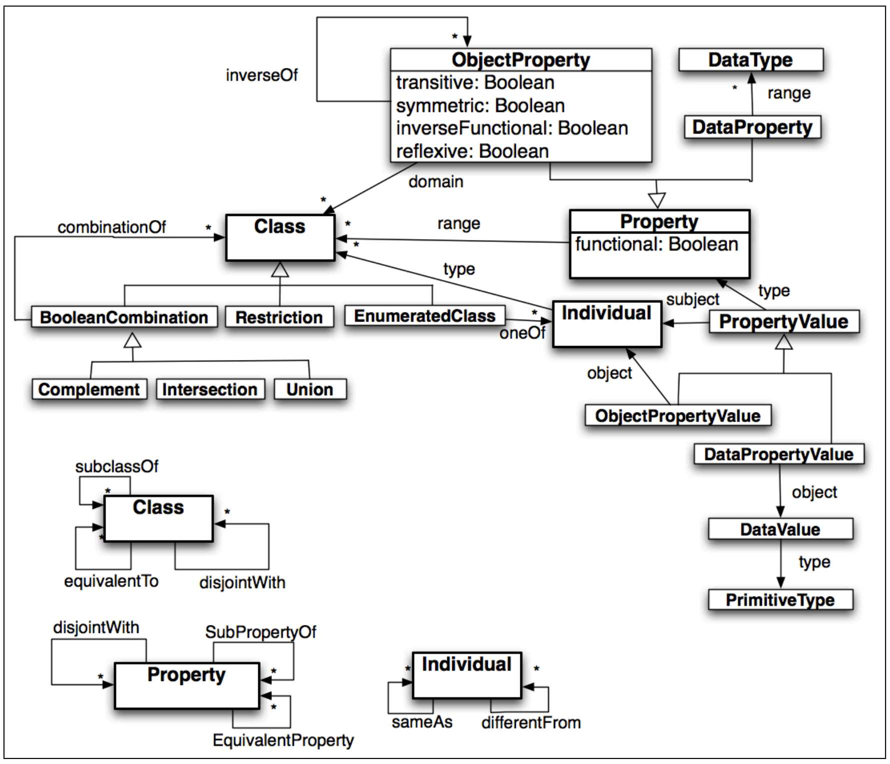


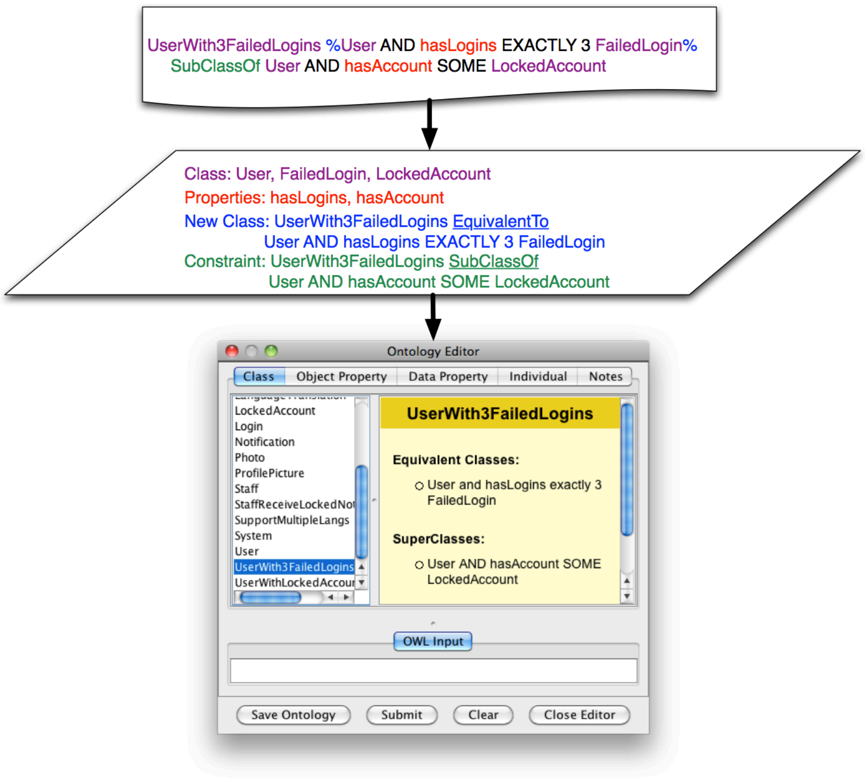
Fig. 1 Ontology's meta-model

3.3.2 How to build and maintain an ontology?

The RE process normally starts with domain studies. Here, requirements engineers gain an understanding of the system domain, environment and the system's stakeholders (van Lamsweerde, 2003). The creation of the ontology should fall within this phase. The ontology creation can be seen as a two-step process: obtaining ontology elements and incorporating domain knowledge into the ontology.

Although building a domain ontology is current out of scope of this work, there are a number of sources and methodologies that could be utilized to obtain ontology entities for specific domains. TONES Ontology Repository (TONES, 2008), Protégé Ontology Library (Wiki, 2007) and others (D'Aquin and Noy, 2011) provide rich sources of ontologies in a variety of domains. These libraries are actively contributed to by many researchers and developers in the semantics web area. In addition, techniques for extracting ontology elements from natural language texts (Goldin and Berry, 1997), ontology learning (Maedche and Staab, 2001) and web-mining (Kaiya et al., 2010) can augment the ontology elements collection process.

The step of incorporating domain knowledge into the ontology is carried out as the understanding of requirements engineers about the domain evolves. Upon dis-



```
UserWith3FailedLogins %User AND hasLogins EXACTLY 3 FailedLogin%
   SubClassOf User AND hasAccount SOME LockedAccount
```

```
Class: User, FailedLogin, LockedAccount
Properties: hasLogins, hasAccount
New Class: UserWith3FailedLogins EquivalentTo
           User AND hasLogins EXACTLY 3 FailedLogin
Constraint: UserWith3FailedLogins SubClassOf
           User AND hasAccount SOME LockedAccount
```

UserWith3FailedLogins

Equivalent Classes:

- User and hasLogins exactly 3 FailedLogin

SuperClasses:

- User AND hasAccount SOME LockedAccount

Fig. 2 A requirement incorporated into the Ontology

covery, domain knowledge is formalized in MOS, and is then translated into a set of ontology elements and incorporated into the ontology.

While the creation of the domain ontology should be carried out at from the beginning of domain study phase, it can be extended at any time during the RE process whenever new knowledge about the domain is obtained.

During the RE process, requirements are formalized and incorporated into the ontology in the same way as domain knowledge. Figure 2 shows an example of this knowledge incorporation process.

The top box shows the formalization of the requirement "*If a user makes 3 failed login attempts, then their account will be locked*." The formalization is then interpreted in order to map the requirement to relevant ontology elements. We identify a number relevant classes and properties in which a new class called `UserWith3FailedLogins` has been defined (The definition of it is: `% User AND hasLogins EXACTLY 3 FailedLogin %`). In addition, the requirement is interpreted in a form of a constraint that the `User` must have an account, which has been locked after three failed login attempts: `UserWith3FailedLogins SubClassOf User AND hasAccount SOME LockedAccount`. The last box shows that the ontology is updated with the new definition and constraint.The entire process of requirement interpretation and ontology update is carried out automatically by our tool REInDetector which will be discussed in section 5.

3.3.3 How to evaluate an ontology's quality?

In KBRE, as the results of requirements analysis and reasoning (see section 4 largely depend on the ontology's quality, it is important that the ontology is ensured to be correct, complete and consistent.

The ontology's correctness and completeness are dependent on the ontology development methodology used, the skills of knowledge engineers, time constraints and the availability of supporting documents and resources. In addition, since it is a general consensus that there is no single way to develop ontologies and no correct ontology for any particular domain (Noy et al., 2001), and with the focus solely on Requirements Engineering, ensuring the ontology's correctness and incompleteness is beyond the scope of this work.

In KBRE the ontology's inconsistency can be guaranteed with the assistance of *REInDetector*, which supports the identification of inconsistencies within the ontology in the same way as how inconsistencies are detected among requirements. The inconsistency detection support will be discussed in more details in section 4.

3.4 Goal Refinement and Operationalization Process

The goal refinement and operationalization process is conducted in both top-down and bottom-up directions. Requirements engineers usually start with high-level goals from stakeholders. They then need to answer HOW questions (i.e. *How can this goal be achieved?*) to refine/operationalize the goal into sub-goals. The sub-goals would be the conditions for satisfying the higher-level goals. On the other hand, requirements engineers may also need to verify the correctness and appropriateness of the refinements/operationalizations by asking themselves WHY questions (i.e. *Why does this goal exist?*). These questions help them verify if the goal is necessary (i.e. *Does it really contribute to satisfying a higher-level goal?*) or if there are other goals needed to be introduced. This bi-directional approach ensures the completeness of the requirements and also avoids redundancy in the goal refinement process.

In our KBRE model, the refinement and operationalization process is assisted by the use of a *goal graph*. The goal graph visualizes how high-level goals are refined/operationalized into lower-level goals, or alternatively, how low-level goals contribute to the realization of higher-level goals. For a particular goal, it can be refined into multiple sub-goals via different types of refinement links including AND-links, OR-links and Optional-links. Each of these link types is described in the following.

3.4.1 AND-Link

In KBRE, AND-Links are used for cases of *minimal refinement*, which means the goal can only be satisfied if all of the sub-goals linked to it via AND-Links are sat-

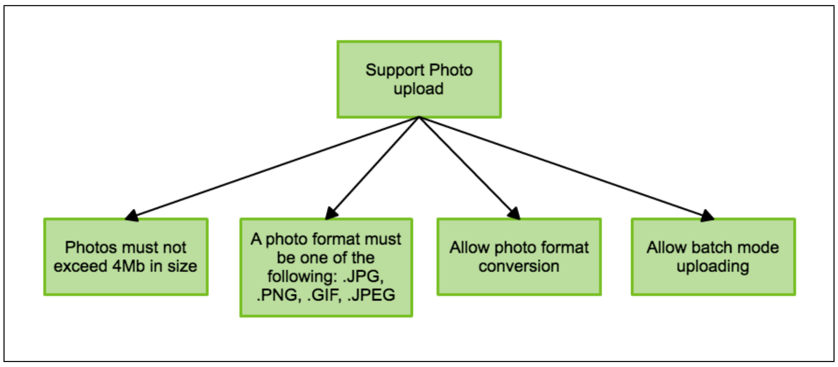

Support Photo upload
Photos must not exceed 4Mb in size
A photo format must be one of the following: .JPG, .PNG, .GIF, .JPEG
Allow photo format conversion
Allow batch mode uploading

Fig. 3 An example of AND-Link

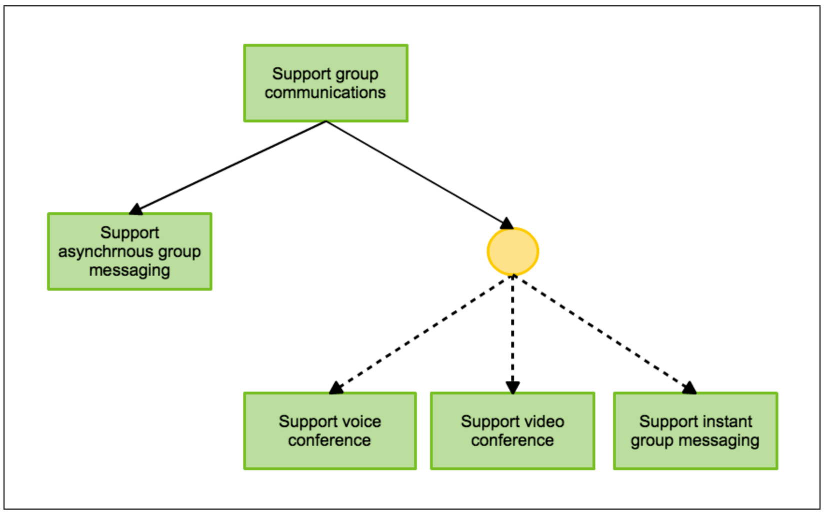

Support group communications
Support asynchrnous group messaging
Support voice conference
Support video conference
Support instant group messaging

Fig. 4 An example of OR-Link

isfied. In other words, if there is any of the sub-goals unsatisfied, the goal will not be fulfilled. An example of an AND-Link is illustrated in Figure 3. In this example, the goal 'Support Photo Upload' is refined into 4 sub-goals, all with AND-Links. That means the satisfaction of that goal can only be achieved if all the 4 sub-goals are fulfilled.

3.4.2 OR-Link

OR-Links are used in cases of *alternative refinement*, which means the goal being refined can be satisfied by fulfilling any of the sub-goals involved in the OR-Links. Conversely, satisfying any of the sub-goals would fulfill the goal. In the model, whenever OR-Links are needed, a virtual goal must be created to link the goal with its alternative refinements. This virtual goal is connected to the goal via an OR-Link and to the sub-goals via AND-Links. Figure 4 illustrates an example of an OR-Link. In

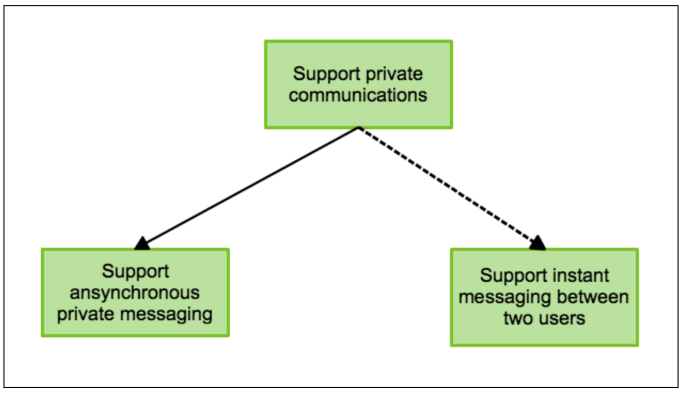

Support private communications
Support ansynchronous private messaging
Support instant messaging between two users

Fig. 5 An example of Optional-Link

this example, OR-Links are used to allow the specification of three alternatives of supporting group communication (any one of the three options plus the goal 'Support asynchronous group messaging' will suffice to satisfy the goal 'Support group communication'). The yellow circle denotes a virtual goal, which is created only for the purpose of linking alternative options to the top goal via OR-Links.

3.4.3 Optional-Link

In the KBRE model, Optional-Links are used in cases of *optional refinement*. This denotes that the sub-goals involved in Optional-Links are the preferred options but they are not strictly required for the higher-level goal to be fulfilled. They may contribute to the realization of the goal being refined. However, without such sub-goals, the goal can still be satisfied. In the goal graph, Optional-Links can be used to connect functional goals or non-functional goals to any other goals. However it can't be used to connect virtual goals to any other goals. Figure 5 presents an example of the use of Optional-Link. This example shows a case of refining the goal 'Support private communications'. According to the refinement, this goal can be achieved by supporting asynchronous private messaging between users in the system. However, it is desirable to enable instant messaging services. This feature is considered a "nice-to-have" feature and thus its existence does not critically affect the satisfaction of the parent goal.

3.5 Goal Annotations

In the KBRE model, each goal is annotated by a number of features that individually characterize the goal. The goal annotation elements can be described as follows (with Figure 6 illustrating an example the annotations of a goal that specifies that the system needs to support private communication between users):

– **Name:** This element uniquely identifies the goal within the goal model. For example, in Figure 6, F52_SysSupPrivateComm is the name of a requirement specifying

Name	F52_SysSupPrivateComm
Value	Connecting
GoalType	Functional
Refines	F37_SysSupComm
RefinementLink	AND
RefinedTo	F53_SysSupAsynMessaging, F54_SysSupInstantMessaging, F55_SysSupVoiceCall
InformalDef	System needs to support private communication
FormalDef	System SubClassOf supportCommunication SOME PrivateCommunication

Fig. 6 An example of goal annotations

that the system needs to support private communication between the users of the system.

- **Value:** Each goal has one or more values. A value represents the perspective, or the top most objective, which derives the goal. For instance, the requirement F52_SysSupPrivateComm has two values, namely 'Collaboration' and 'Social Activity'. This indicates that the added value to the system, once this requirement is fulfilled, is to enable collaboration between users and social activities for users.
- **GoalType:** each goal has a type, which can be '**Functional**', '**Non-functional**' or '**Mixed**', when the requirement can contribute to both functional and non-functional features of the system.
- **Refines:** This element indicates the higher-level goal being refined. In other words, it refers to the parent goal of the annotated goal. Unless this is the top-most goal, which doesn't refine any other goal, this element refers to a unique parent goal. To deal with cross-cutting sub-goals that have multiple parent goals, we are currently extending our model so that goal graphs can be arbitrary graphs rather than just a tree. The extension however makes the analysis and reasoning about goals much more complex.
- **Refinement Link:** This element indicates the type of the refinement link from the annotated goal's parent to itself (this is empty if the goal is the top most one). According to the previous section, the refinement type can be AND, OR or Optional.
- **RefinedTo:** This element indicates the list of goals which are generated as the results of the refinement on this goal. We refer to them as the 'children' goals of the annotated goal.
- **InformalDef:** This defines the goal in natural language. For instance, '*The system needs to support private communication*'.
- **FormalDef:** This defines the goal in the formal requirements specification language. For instance, the above goal is formally defined as: `'System` **`SubClassOf`** `supportCommunication` **`SOME`** `PrivateCommunication'`.

Figure 6 illustrates the annotations of the goal specifying that the system needs to support private communications between users. It comes from the refinement of a higher-level goal requiring that the system allow methods of communications (*Refines*). The goal is derived from the objective of encouraging social activities as well as work collaborations among users (*Value*). It is refined into 3 other sub-goals, which are F10_SysSupAsynMessaging, F11_ SysSupInstantMessaging and F12_SysSupVoiceCall.

4 Analysis and Reasoning Support

Apart from detecting inconsistencies between requirements, our model also provides several mechanisms for identifying redundancies and overlaps of requirements. In the rest of the paper, we use the term *"requirements problems"* to refer to these three issues. In this section, we describe the analysis and reasoning services for identifying such requirements problems. We assume a common knowledge base that stores all relevant information produced during the requirements engineering process, including the ontology and all of the requirements specifications. While we allow the requirements engineer to formally specify requirements using the Manchester OWL syntax (MOS), there is a one-to-one translation mapping MOS statements to description logic axioms. Since the assertions expressing the ontology can also be represented as description logic axioms, the whole knowledge base consists of a set of axioms in the *SROIQ* logic.

4.1 Inconsistency Detection

From a requirements engineering perspective, a set of requirements is consistent if their specifications do not contain any contradiction and they are not in conflict with each other. Inconsistencies are the impossibility in satisfying requirements which may arise from competing objectives. For instance, If a requirement is derived from a stakeholder with the purpose of increasing usability of the system; it is likely to be in conflict with another requirement put forward by a stakeholder with the objective of saving development cost. In our model, as all requirements and domain knowledge are captured in a description logic knowledge base, we can appeal to the logical consistency of the knowledge base to formally define the notion of consistency (and inconsistency) between requirements.

Inconsistency arises in a description logic knowledge base when there are two or more axioms of the knowledge base that cannot concurrently exist. Formally,

Definition 1. A set of axioms $\{A_1, A_2, \ldots A_k\}$ is inconsistent if and only if there does not exist any interpretation that concurrently makes the axioms $A_1, A_2, \ldots, A_k$ true.

In the description logic *SROIQ*, inconsistency can be explained as a situation in which a concept or an object cannot exist. From the definition of inconsistency, we define the minimal set of inconsistent axioms, called *minimal set of inconsistency*. Informally, a minimal set of inconsistency is an inconsistent set of axioms and the inconsistency can be solved by removing any one of the assertions.

Definition 2. An inconsistent set of axioms $\{A_1, A_2, \ldots, A_k\}$ is a minimal set of inconsistency if and only if for each $i = 1, \ldots, k$, the set of axioms $\{A_1, A_2, \ldots, A_k\} \setminus \{A_i\}$ is consistent.

In this work, we employ the description logic reasoner *Pellet* to facilitate several analysis services, including detection of inconsistencies. Specifically, if the reasoner is able to derive that there is an unsatisfiable concept or non-existent instance in the knowledge base then inconsistencies are identified. An example of an inconsistency is given in the following example:

Example 3. Consider the following security requirements, presented both in the form of natural language and in the corresponding MOS specification:

R_1: After three continuous failed login attempts, the account would be locked by the system (`User AND hasFailedLogins VALUE 3 SubClassOf User AND hasAccount SOME LockedAccount`).

R_2: Once an account is locked, the system sends an account lock notification email to the account's owner (`User AND hasAccount SOME LockedAccount SubClassOf User AND receiveEmail SOME AccLockedNotification`).

R_3: Once an account is locked, the system would also send a SMS message to the account's owner to notify him/her about the situation owner (`User AND hasAccount SOME LockedAccount SubClassOf User AND receiveSMSMessage SOME AccLocked-Notification`). And another requirement with the goal of minimizing unnecessary features:

R_4: If a user already received a notification via email, he/she won't receive the same notification in SMS (`User AND receiveEmail SOME Notification SubClassOf Not (User AND receiveSMS SOME Notification)`).

Based on the ontology that holds the definitions of the concepts and properties in the above formalization, we are able to detect that this is an inconsistency. We can also determine that the minimal inconsistent requirement set contains R_2, R_3 and R_4 by pointing out that the class of `(User AND has Account SOME LockedAccount)` is unsatisfiable and the removal of any of the three requirements would eliminate the inconsistency.

4.2 Detecting Redundancies

Definition 3. A description logic knowledge base is said to be redundant if it contains an axiom that can be inferred from the other axioms in the knowledge base.

In the *SROIQ* logic, redundancy can be explained as a situation in which an inclusion axiom between two concepts or between an instance and a concept is redundant. For instance, if the knowledge base contains the axioms: $C \sqsubseteq D$, $D \sqsubseteq E$, and $C(a)$, then the axioms $C \sqsubseteq E$ and $D(a)$ are both redundant. Axiom $C \sqsubseteq E$ can be derived from $C \sqsubseteq D$ and $D \sqsubseteq E$, while axiom $D(a)$ can be derived from $C(a)$ and $C \sqsubseteq D$. These are two basic inference problems in description logic, namely the *subsumption* and the *instantiation* problems (Baader et al., 2007).

In RE, redundancies are not always considered serious problems as their existence normally does not affect the correctness of the requirement set. They are instead referred to as warnings to requirements engineers as they may cause confusion or unnecessary documentation effort. Moreover, the elimination of redundancies could potentially lead to significant optimizations in requirement analysis and reasoning.

In our approach, identification of redundancies is performed by the DL reasoner Pellet by reducing the inference problems subsumption and instantiation to checking consistency of a knowledge base. For instance, to check whether $C \sqsubseteq E$ follows from a knowledge base $\mathcal{K}$, we can verify whether $\mathcal{K} \cup \{C \sqcap \neg D\}$ is inconsistent. The following example demonstrates a case of requirement redundancy.

Example 4. There are three requirements regarding the supported languages in a Social Networking System as follows:

R_1: The system needs to support all three languages: English, French and Japanese (`System SubClassOf supportLanguage VALUE {English, French, Japanese}`).

R_2: The system needs to support language translation (`System SubClassOf supportFeature VALUE LanguageTranslation`).

R_3: If supporting multiple languages, the system needs to allow language translation (`System AND supportLanguage MIN 2 SubClassOf supportFeature VALUE LanguageTransaltion`).

In this example, R_2 is redundant because the fact that the system needs to support language translation can be inferred from R_1 and R_3. Indeed, the system is required to support three languages. That means it support multiple languages and according to R_3, the feature of language translation needs to be included in the system.

4.3 Detecting Overlaps

Overlaps refer to situations in which there are two or more assertions that refer to some common or inter-related phenomena. Overlap and inconsistency are considered two levels of interference between specifications; the first is the pre-requisite for the second (DeMarco, 1979). Overlaps normally arise from differences in stakeholders' preferences or opinions in regard to what should be done to satisfy a certain aspect of a system.

In the *SROIQ* logic and OWL language, an overlap can be explained as a situation in which there are at least two assertions that contain the same concepts (aka. classes in OWL language) or instances (aka. individuals in OWL language).

In Requirements Engineering, overlaps occur when there are two or more requirements specifying constraints on the same features, or rules. Therefore, overlaps can lead to inconsistencies if one of these requirements changes and causes conflicts to others. Subsequently, overlaps are not considered serious problems but instead are flagged as warnings for potential sources of inconsistencies in requirements.

Based on the above description, in the KBRE model, overlaps are detected by checking the subsumption relationships of the related classes in the ontology (two or more classes are 'related' if they have subsumption relationship or share at least one instance). The following example illustrates an occurrence of overlaps.

Example 5. Given an instance `'John'` that belongs to both `Student` and `Staff` classes. The two requirements below are overlapped in the context of a library management system:

Problem Inconsistency

Explanation UserWithLockedAccount class is unsatisfiable

Problematic Requirements F62_IfLockedThenSMS (Security), F63_IfLockedThenEmail (Security), F71_IfEmailThenNoSMS (Cost-effective)

Reason(s)

```
UserWithLockedAccount SubClassOf User AND receiveSMS SOME
AccLockedNotif

UserWithLockedAccount SubClassOf User AND receiveEmail SOME
AccLockedNotif

UserReceiveLockedNotifEmail SubClassOf NOT User AND receiveSMS
SOME AccLockedNotif

UserReceiveLockedNotifEmail EquivalentTo User AND receiveEmail SOME
AccLockedNotif
```

Fig. 7 An example of explanations

R_1: Staff are allowed to keep their borrowed items up to 30 days (`Staff SubClassOf hasRight SOME (Right AND keepBook MAX 30)`).

R_2: Students are allowed to keep their borrowed items up to 20 days (`Student SubClassOf hasRight SOME (Right AND keepBook MAX 20)`).

R_1 and R_2 are overlapped because they specify the same rule (maximum loan duration) on two related sets of actors (`Student` and `Staff`) that have an instance (`'John'`) in common.

4.4 Explanations

For any detected requirements problems, detailed explanations are provided to help requirements engineers and their stakeholders get better insights into the problem. The explanations include the main reason why the problem exists (e.g., which classes are in conflict or which instance must not exist,... and why) and the "hypothesis" underlying the problem (e.g., a requirement whose value is 'cost-effective' can be inconsistent with a requirement that aims to improve 'usability' through some complex user interface features).

Figure 7 shows the explanations for the inconsistency between the security requirements in example 4. In this example, the three requirements F18_IfLockedThenSMS, F17_IfLockedThenEmail, F19_IfEmailThenNoSMS are inconsistent because they lead to the fact that the class `UserWithLockedAccount` is unsatisfiable. According to the explanations, this problem is due to the conflicting between the *Security* and *Cost-effective* goals. The causes of that are presented under the **Reason(s)** part of the explanations.

4.5 Querying Requirements

When a set of requirements is detected as having potential requirements problems (i.e., inconsistency, redundancy or overlap), it is sometimes necessary for requirements engineers to take a closer look to get more insights into the requirements, understand the problems and probably try to resolve them. For instance, if inconsistency is detected in a knowledge base with hundreds of requirements, it is not likely that all of these requirements are involved in causing the conflict. There may be only two or three requirements directly involved in the problem. In these situations, a query for minimal set of inconsistent requirements would enable the analysis of the inconsistency more effectively. Furthermore, requirements engineers may only be interested in a subset of requirements rather than all requirements for the entire system. For instance, they only need to study the requirements relevant to the communication features in a social networking system. To address these needs, the KBRE model allows requirements engineers to query particular subsets of requirements and for the specific types of problems they are concerned about. In particular, they can choose a specific list of requirements and check for inconsistencies or redundancies among the selected set of requirements.

5 The REInDetector Tool

We have developed *REInDetector* - a prototype tool to demonstrate the feasibility of our new goal-directed, DL-based RE model as an automated RE framework. *REInDetector* allows requirements engineers to enter requirements, specify relationships between requirements (via AND, OR and Optional links), and formalize requirements in the Manchester OWL Syntax. The tool also supports the creation and maintenance of an ontology for storing the domain knowledge and semantics of requirements. *REInDetector* equips requirements engineers with the analysis services described in Section 4. The analysis services provided in *REInDetector* are largely based on the Pellet reasoner (Clark & Parsia, LCC, 2012), which is extended to facilitate the identification of redundancies and overlaps and to allow more comprehensive explanations for detected requirements problems (including inconsistencies, redundancies, and overlaps).

Figure 8 illustrates the high-level architecture of our *REInDetector* tool, consisting of three main modules. The *Input module* is responsible for taking specifications from requirements engineers. The *Requirement Elicitor* enables one to capture a system's functional and non-functional requirements and shows how they are related through refinement links. Knowledge, semantics, rules, and constraints in the requirements domain can be defined using the *Ontology Editor*, which allows users to directly interact with the ontology. All these types of inputs need to be properly formalized in order to allow the effective analysis and reasoning on requirements. The formalizations of inputs are then sent to the *Manchester OWL Syntax parser* (*MOS parser*) in the *Knowledge module*. This parser then extracts the classes, roles, instances ,and the relationships declared in the formalization and sends the data to the *Ontology manager* which then updates the ontology with new data. The *Reasoner*

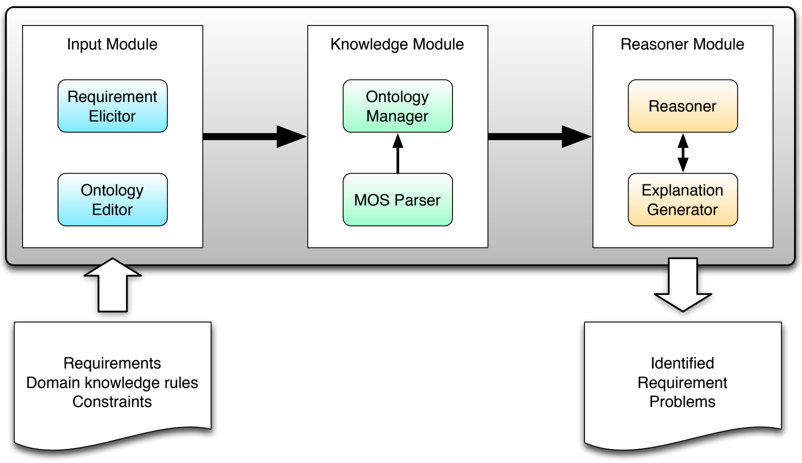


Fig. 8 General structure of *REInDetector*

module is responsible for providing analysis and reasoning services, including detection of requirements problems, giving answers to requirements queries (carried out by the *Reasoner* component) and generating explanations (carried out by the *Explanation Generator*).

REInDetector is a Java application that provides a graphical user interface (based on the JUNG library (O'Madadhain et al., 2005)) to perform all requirements elicitation, management, and validation tasks (Zave and Jackson, 1997). We have developed a dedicated structuring and management system to capture requirements and their relationships following the Goal-oriented Requirements Engineering model (Van Lamsweerde, 2001). In addition, the use of Description Logic (Baader et al., 2007) enables us to define object/class-style ontologies, which are the core for requirements formalization and analysis in *REInDetector*. Description Logic specifications can be faithfully mapped to Manchester OWL syntax (Horridge and Patel-Schneider, 2009). As a result, we are able to use the off-the-shelf OWL 2 reasoner Pellet (Clark & Parsia, LCC, 2012) for requirements analysis. However, rather than using an existing OWL editor such as Protégé (Noy et al., 2003) or SWOOP (Kalyanpur et al., 2006), which are too general-purpose, we constructed our own ontology editor in order to obtain a better match with the Requirements Engineering domain.

REInDetector finds implicit consequences of explicit requirements and offers all stakeholders an additional means to identify problems in a more timely fashion than existing RE tools. An important feature of *REInDetector* is its ability to generate comprehensive explanations to provide deeper insights into the detected inconsistencies. We rely on Pellet for inconsistency checking and explanation generation. However, in order to detect redundancies and overlaps we also had to implement a number of subsumption relationship verifications so that these requirements issues can be checked and explained also.

6 Evaluation

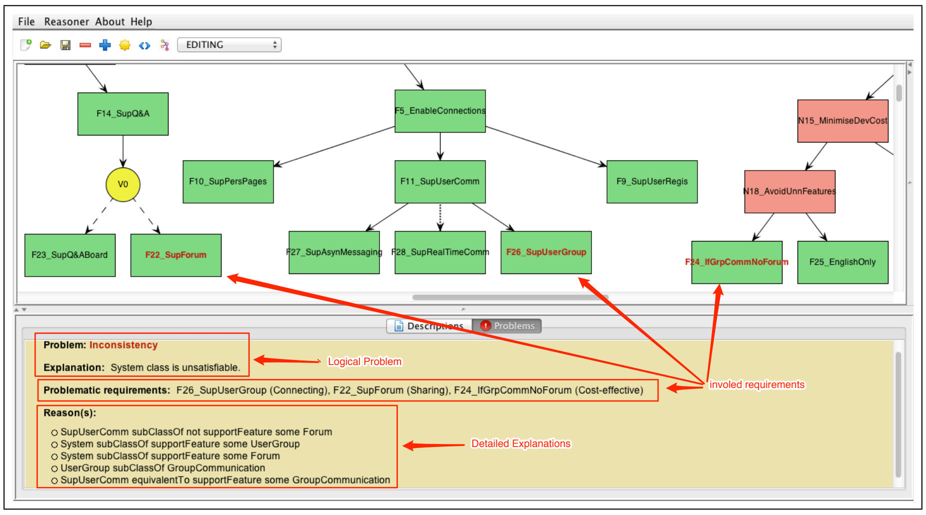
To evaluate the effectiveness of our KBRE model and proof of concept tool, we have used a complex, industrial software development scenario for design and development of a traveler social networking system and verified the results produced by *REInDetector*. Our motivation for choosing this scenario was twofold. First, it involves the development of a large-scale software system with many diverse components. This enabled us to test our KBRE model on a variety of varying requirement types. Second, due to the size of the system, requirements originate from different stakeholders perspectives that may naturally give rise to emerging inconsistencies. We selected a subset of approx. 80 requirements from the "*Review and Rating*" and "*Blogging*" components for evaluation, as these components constitute the main building blocks in the system. The selection of these requirements was based on our awareness of the potential conflicts induced by the competing `Usability`, `Cost-effective`, and `Reliability` goals. We opted to conduct this type of evaluation instead of carrying out an end user survey on our Knowledge-based RE approach, as our focus was on the applicability and utility of the KBRE model.

This case study comprises an ontology with more than 350 classes, properties, and instances. *REInDetector* found all inconsistencies, redundancies, and overlaps in the requirement set, manually verified by us. The inconsistencies arose from logical conflicts between requirement goals. It took about 4 seconds to detect inconsistencies, 9 seconds to locate the redundancies, and 14 seconds to identify all overlaps in the selected requirements set. The experiment was carried out on a MacBook Pro running MacOS X 10.6.8 with a 2.53 GHz Intel Core 2 Duo processor and 4Gb of 1067 MHz DDR3 Memory. The running time for detecting redundancies and overlaps are significantly longer than that for checking inconsistencies, because redundancies and overlaps need to be first reduced to a knowledge base consistency problem, which usually requires multiple queries to be posed to the reasoner. Considering that redundancies and overlaps are not the most serious issues in requirements engineering and that they are, in general, not as frequent as inconsistencies, we think that this performance of *REInDetector* is satisfactory.

In this case study, we were able to detect complicated inconsistencies involving multiple requirements. The requirements mostly came from conflicting viewpoints among stakeholders (e.g., `Cost-effective` vs. `Security`, `Cost-effective` vs. `Usability`, `Usability` vs. `Functionality`, etc.). Figure 9 presents an annoted screenshot of the *REInDetector* system in which inconsistency is detected and the explanations for it are also provided.

This scenario focuses on the decision of connecting travellers and providing a place for them to come to for discussions, exchanging experiences and seeking advices/tips for their coming trips. There are two requirements promoting the user experience with the system:

- F22_SupForum: System needs to provide a forum for discussions (`System SubClassOf supportFeature SOME Forum`).
- F26_SupUserGroup: System supports the creation of user groups to facilitate the connection (`System SubClassOf supportFeature SOME UserGroup`).



Problem: Inconsistency

Explanation: System class is unsatisfiable.

Problematic requirements: F26_SupUserGroup (Connecting), F22_SupForum (Sharing), F24_IfGrpCommNoForum (Cost-effective)

Reason(s):

- SupUserComm subClassOf not supportFeature some Forum
- System subClassOf supportFeature some UserGroup
- System subClassOf supportFeature some Forum
- UserGroup subClassOf GroupCommunication
- SupUserComm equivalentTo supportFeature some GroupCommunication

Fig. 9 Example of an identified inconsistency

However, stakeholders with the `Cost-effective` point of view had another requirement with the aim of minimizing the development cost:

– F24_IfGrpCommNoForum: If the system supports any type of group communication, then no forum is needed (`supportFeature SOME GroupCommunication NOT (SubClassOf supportFeature SOME Forum)`).

The underlying rationale is that, with any group communication feature, travelers would be able to contact and discuss in a group, therefore, it would be a waste to implement a forum.

We found this case interesting as `UserGroup` is a type of `GroupCommunication`, and the conflict among the three requirements can't be identified without having the relationships between `UserGroup` and `GroupCommunication` explicitly defined.

The corresponding explanations provide a rationale governing the problem. Depending on the `Usability` and `Cost-effective` preferences, there are two possible solutions to rectify the problem. First, if the viewpoint `Usability` is more important than `Cost-effective`, then a resolution is to remove F24_IfGrpCommNoForum, which would allow users to have more comforts in participating in discussions and shadings as a forum is far better than user groups in discussions due to the better organization and structure. Alternatively, we can remove F22_SupForum or F26_SupUserGroup, which would also solve the issue, but at the expense of `Usability`. Without a forum or user groups, travelers may not be provided with the best facilities to connect with each other and exchange experiences and thus the satisfaction of the project's objectives may not be guaranteed. The decision here must be made by the stakeholders, probably through some kinds of negotiation.

Nevertheless, while this case study *REInDetector* was able to identify all problems related to the requirements expressible in Manchester OWL Syntax (MOS) as long as the relevant concepts and roles are completely specified in the ontology, the tool is not able to detect the conflicts associated with the requirements that are not expressible in MOS. This is expected since there are a number of limitations with MOS expressiveness which prevent certain types of requirements to be captured. For instance, it is not possible to use MOS for representing requirements with temporal properties. Thus, a requirement such as *"When the user chooses to show his/her online status, the user's status button will always reflect the user's availability on the system"* can not be expressed in MOS.

7 Related Work

The related works to our approach can be categorized into two groups; one is ontology formal representation languages which allow automated analysis and reasoning and another include the approaches for detecting inconsistencies in RE. The formal languages proposed in the community of knowledge, ontology and requirements engineering can be classified into two categories: languages based directly on logics and controlled subsets of English which is translatable to logics for reasoning. The works fall into the first category include: RML (Greenspan et al., 1994), TELOS (Mylopoulos et al., 1990), FRORL (Tsai et al., 1992), KAOS requirements representation language (Dardenne et al., 1993; Van Lamsweerde, 2001), Formal Tropos (Fuxman et al., 2004). The common problem with these languages is that although being very powerful representation languages with the expressiveness inherited from logics, they are fairly complicated and normally require technical training for their use and elaboration (Jaramillo et al., 2006).

In addition, Controlled Natural Languages (CNLs) has been a recently emerging area. Research in this field has attempted to overcome the limitations of traditional logic-based representation languages by proposing different sub-sets of English which are transformable to logics or formal languages to allow automated analysis and reasoning. Some examples of well-known CNLs include ACE (Fuchs et al., 2005), PENG Schwitter (2002), V2E (Pratt-Hartmann, 2003) and SOS (Cregan et al., 2007). ACE, PENG and V2E are transformable to First-order Logic (FOL), represent a great combination of Logics and natural languages and thus are powerful in both expressiveness and user-friendliness. However, except V2E which is proved to be equivalent to the decidable 2-variable sub-set of FOL (in term of expressiveness), other languages are known to be undecidable. Moreover, although possessing great expressiveness power, these languages are proved not suitable to represent concepts, roles and their relationships, due to their syntactic constraints (inherited from English grammar) (Schwitter et al., 2008). In particular, ACE neither provides explicit constructs for enumerations nor allows datatype properties to be defined. In addition, representing relationships between roles or specifying disjoints entities are very complex in ACE.

SOS is the language most comparable to MOS. It is a controlled natural language which is transformable to OWL1.1 to allow the representation of OWL ontologies and automated reasoning. The disadvantage of SOS is that it is equivalent to OWL1.1 while MOS is equivalent to OWL2 which is more expressive. More precisely, OWL2 better supports qualified number restrictions, property and data type expressivity.

MOS offers us a good option for balancing between the user-friendliness, efficiency and expressiveness of a representation language. MOS, with its expressions constructed from natural language words, has been proved to be well-received by non-logicians (Cregan et al., 2007). In addition, being transformable to Description Logics, a decidable sub-set of FOL, MOS guarantees the sound and complete reasoning. Moreover, using MOS is also benefited from the availability of various reasoners for Description Logics. Furthermore, although possessing less expressive power than FOL or Temporal logics (MOS is not able to allow temporal operators), MOS is powerful in its representation of domain knowledge, with concepts, roles, and instances and their relationships, the key focus of our work here.

A main limitation of MOS is that it is not capable of representing temporal constraints. For instance, a requirement such as *"When the user chooses to show his/her online status, the user's status button will always reflect the user's availability on the system"* can not be expressed in the language.

In the next part of this section, we discuss the key related research approaches that have been proposed for dealing with the problem of detecting requirements inconsistencies in RE.

WinWin (Boehm et al., 1995) is a human based-collaborative approach designed for detecting conflicts between stakeholders' win conditions. This approach doesn't provide explicit support for conflict detection. It and its QARCC extension (Boehm and In, 1999), instead allow stakeholders to enter win conditions and these conditions are associated with quality attributes. Based on the predefined knowledge base of potential conflicting quality attributes, potential conflicts between win conditions are flagged. Although this approach can produce fast and precise results in some situations, it may not be complete and sound because the correctness of the identification process heavily depends on the completeness of the knowledge base. In addition, it is labour-intensive and difficult to be applied for large models.

Similar to WinWin, Robinson and Pawlowski (1999) proposed a technique called DealScribe following the a human-based collaborative approach. In DealScribe, stakeholders are expected to identify conflicts between the root requirements, which are the most general requirements defined for the relevant concepts in the models. The stakeholders are expected to explore and indicate the interactions between all the possible pairs of root requirements in the models. An interaction may be characterized as "very conflicting," "conflicting," "neutral," "supporting," or "very supporting." The conflicts among requirements in the models would then be detected based on these interactions. This technique has similar drawbacks to those of WinWin.

Sharing the same idea of predefining potential conflicting software quality attributes with WinWin and DealScribe, Egyed and Grunbacher (2004) also introduce the concept recovering dependencies among requirements. This facilitates the incremental exploration of conflict identification based on the automated traceability of the dependency as the requirements evolve. Although this approach avoids the time-intensive and error-prone process in the other two approaches, its soundness could be challenged as the identification of conflicts still heavily relies on the predefined potential conflicts among quality attributes. In addition, neither the use of domain knowledge and semantics of requirements nor requirements conflicts explanations are supported in this approaches.

KAOS (Dardenne et al., 1993; Van Lamsweerde, 2001) is a comprehensive framework aimed at supporting the process of Requirements Engineering including the management of conflicts among requirements. In KAOS, requirements are formalized in linear temporal logic (LTL) to enable the identification of conflicts using formal reasoning methods. Three of such techniques are described in (Van Lamsweerde et al., 1998) including using regressing negated assertions, divergence patterns and divergence identification heuristics. In addition, other types of conflicts such as process-level deviations or instance-level deviations can also be detected using consistency rule checks.

KAOS provides a very rigorous approach for detecting conflicting requirements. However, as mentioned above, the use of LTL in specifying requirements requires requirements engineers to be sufficiently familiar with LTL and be able to correctly specify requirements in LTL. Thus, its applicability in practice can be fairly limited (Dwyer et al., 1998). In addition, although KAOS supports the creation and maintenance of domain knowledge through the use of its conceptual meta-model, the concepts, roles, and instances in the problem domain as well as the semantics of requirements cannot be adequately expressed and reasoned in LTL (Breaux et al., 2008).

Tropos (Fuxman et al., 2004) is a framework that provides formal and mechanized analysis of early requirement specifications. The requirements analysis in Tropos is done using the model checking approach with the support of a specification language called Formal Tropos which is developed from the combination of the $i*$ modeling language (Chung, 1993) and the temporal specification language inspired by KAOS. Similar to the specification language in KAOS, the Formal Tropos is also too complicated to be used in practice. In addition, the semantics of requirements cannot be defined sufficiently in Tropos as the framework uses no sub-classing or specification of structures and thus the domain and instance-level are not distinguished in the formalization (Breaux et al., 2008). Hence, explanations of conflicts are not provided in both frameworks.

Lauenroth and Pohl (2008) proposed an approach for dynamically checking consistencies among domain requirements in Product Line Engineering. The detection of inconsistencies in their work is based on the combination of the *variability model*, which is for representing the variances among different products in the same product line, and the domain requirements specification model. The dynamic check for inconsistencies is accomplished by using a *contradiction function* which conforms to the model checking methods. Yet, explanations for conflicting requirements as well as domain knowledge and semantics of requirements are not supported.

We are aware of a few works in the area of Ontology-based Requirements Engineering that are close to our approach. Siegemund et al. (2011) proposed building an ontology to support the RE process. However, their focus was mainly on tracking the interactions among requirements (*e.g.*, supporting, retracting, etc.). Inconsistency detection is for identifying conflicts between these interactions rather than between the requirements themselves. In addition, Kaiya and Saeki (2005) also use ontology to define semantics in requirements' domain. However, they translate concepts in requirements into UML class diagrams in which each class represents a concept and introduce different types of links to represent relations among these classes. Based on these some inconsistencies can be detected. The process of translation is omitted in this work. The disadvantage of this approach is that it is not able to express some complex requirements which can be done with Manchester OWL Syntax (*i.e.*, conditioning requirements such as *"a user has 3 failed logins attempt, their account will be locked"*). Moreover, modeling semantics with UML class diagrams would cause complexity when the number of requirements evolves.

Although sharing the same objectives of detecting inconsistencies in RE, the works of Grundy et al. (1998), Kamalrudin et al. (2010) and Weston et al. (2009) have a different focus. Grundy et al. (1998) address the problem of inter-model consistency in software development environment tools. They focus on managing inconsistencies in multiple views which are derived from a common repository. The automated identification of inconsistency is facilitated based on representing the artifacts in the repository by using a special kind of graph. Very limited explanation of the inconsistency root cause is provided. Kamalrudin et al. (2010) focus on detecting inconsistencies between specifications at different levels of abstraction while Weston et al. (2009) propose an approach for checking inconsistencies among semantics compositions in aspect-oriented requirements. As these use semi-formal representations, more limited inconsistency and overlap detection can be achieved than in our description logic-based approach.

To summarize, there have been a number of different approaches sharing common objectives with our work. The key differentiation between the existing approaches and our model lies in the ability of our framework to store domain knowledge and semantics of requirements which assist requirements engineers not only in elaborating requirements but also in identifying requirements problems. Moreover, explanations for conflicting requirements are an integral part of our RE process model allowing requirements engineers to have a better understanding into the nature of the detected problems. Finally, our approach and our requirements specification language permit requirements to be formalized in a simple and intuitive syntax, making it easy for practitioners to understand and communicate between each other.

8 Conclusion

In this paper, we described a new Knowledge-based Requirements Engineering framework (KBRE) aimed at providing a more effective requirements analysis process that also incorporates support for the detection and explanation of inconsistencies, redundancies, and overlaps within a given set of requirements. In the KBRE model, domain knowledge and semantics of requirements are centralized in the form of an ontology while requirements are expressed in an intuitive requirements specification language based on the Manchester OWL Syntax (MOS). Both, the ontology and the requirement specifications, are directly translated into the description logic *SROIQ* that provides a logical and reasoning system to yield a sophisticated, yet practical, means to unveil inherent problems in a given requirements set. Our approach allows the sources of these problems to be traced along the requirements goal graph that results from the application of the Goal-oriented Requirements Engineering (GORE) method to requirements elicitation. Automatically generated explanations can guide requirements engineers towards feasible conflict resolution methods in case of conflicting requirements.

Our Knowledge-based Requirements Engineering method has produced some promising results. The Manchester OWL Syntax (MOS) is sufficient for representing a wide range of requirements. Yet, there are certain features that cannot be formalized in MOS. For example, we are currently unable to formalized temporal properties, as MOS is not equipped to capture those aspects. Moreover, in the KBRE model, the requirements problems can only be detected if the related concepts, roles, and instances as well as their relationships are declared in the ontology. As a consequence, specific requirements problems may escape detection if ontology is not powerful (*i.e.*, descriptive) enough. Given the limitation of OWL expressiveness in dealing with temporal operators, we intend to extend the requirements specification language with temporal constructors. Significant upgrades would then be expected in the underlying reasoner to cope with the extensions of the specification language. In addition, we are planning to work on an automated mechanism to produce warnings to requirements engineers in the situations when potential concepts, roles, instances, and their relationships are not declared in the ontology. Finally, more comprehensive evaluation will need to be carried out in order to assess the effectiveness of our framework in varying problem domains and detailed requirements engineer feedback on our approach and toolset.

References

Anton, A. (1996). Goal-based Requirements Analysis. In *Proceedings of the Second International Conference on Requirements Engineering*, pages 136–144. IEEE.

Baader, F., Horrocks, I., and Sattler, U. (2007). Description logics. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*. Elsevier.

Boehm, B., Bose, P., Horowitz, E., and Lee, M. (1995). Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach. In *Proceeding of the 17th International Conference on Software Engineering*, pages 243–243. IEEE.

Boehm, B. and In, H. (1999). Conflict Analysis and Negotiation Aids for Cost-Quality Requirements. *Software Quality Professional*, 1(2):38–50.

Breaux, T., Antón, A., and Doyle, J. (2008). Semantic Parameterization: A Process for Modeling Domain Descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):5.

Brockmans, S., Volz, R., Eberhart, A., and Löffler, P. (2004). Visual modeling of owl dl ontologies using uml. *The Semantic Web–ISWC 2004*, pages 198–213.

Chung, K. (1993). Representing and Using Non-functional Requirements: A Process-oriented Approach. *IEEE Transactions on Software Engineering*, 18(6):483497.

Clark & Parsia, LCC (2012). Pellet - OWL 2 Reasoner.

Corcho, O. and Gómez-Pérez, A. (2000). A Roadmap to Ontology Specification Languages. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, EKAW ’00, pages 80–96, London, UK. Springer.

Cregan, A., Schwitter, R., and Meyer, T. (2007). Sydney OWL Syntax - Towards a Controlled Natural Language Syntax for OWL 1.1. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*. OWLED.

D'Aquin, M. and Noy, N. (2011). Where to publish and find ontologies? A survey of ontology libraries. *Web Semantics: Science, Services and Agents on the World Wide Web*.

Dardenne, A., Fickas, S., and van Lamsweerde, A. (1991). Goal-directed Concept Acquisition in Requirements Elicitation. In *Proceedings of the 6th International Workshop on Software Specification and Design*, pages 14–21. IEEE Computer Society Press.

Dardenne, A., Van Lamsweerde, A., and Fickas, S. (1993). Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2):3–50.

DeMarco, T. (1979). *Structured Analysis and System Specification*. Yourdon Press.

Doe, R. (2009). The standish group chaos report.

Dwyer, M., Avrunin, G., and Corbett, J. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1998). Property Specification Patterns for Finite-State Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15.

Egyed, A. and Grunbacher, P. (2004). Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. *IEEE Software*, 21(6):50–58.

Fuchs, N., Höfler, S., Kaljurand, K., Rinaldi, F., and Schneider, G. (2005). Attempto Controlled English: A Knowledge Representation Language Readable By Humans and Machines. *Reasoning Web*, pages 95–95.

Fuxman, A., Liu, L., Mylopoulos, J., Pistore, M., Roveri, M., and Traverso, P. (2004). Specifying and Analyzing Early Requirements in Tropos. *Requirements Engineering*, 9(2):132–150.

Goldin, L. and Berry, D. (1997). Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. *Automated Software Engineering*, 4(4):375–412.

Greenspan, S., Mylopoulos, J., and Borgida, A. (1994). On formal requirements modeling languages: Rml revisited. In *Proceedings of the 16th international conference on Software engineering*, pages 135–147. IEEE Computer Society Press.

Grundy, J., Hosking, J., and Mugridge, W. (1998). Inconsistency Management for Multiple-view Software Development Environments. *IEEE Transactions on Software Engineering*, 24(11):960–981.

Guarino, N., Oberle, D., and Staab, S. (2009). What is an ontology? *Handbook on Ontologies*, pages 1–17.

Henderson, P. (2006). Why Large IT Projects Fail. *ACM Transactions on Programming Languages and Systems*, 15(5):795–825.

Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., and Wang, H. (2006). The Manchester OWL Syntax. *OWL: Experiences and Directions*, pages 10–11.

Horridge, M. and Patel-Schneider, P. F. (2009). OWL 2 Web Ontology Language Manchester Syntax. W3C Working Group Note, W3C.

Horrocks, I., Kutz, O., and Sattler, U. (2006). The Even More Irresistible SROIQ. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 57–67.

Jaramillo, C., Gelbukh, A., and Isaza, F. (2006). Pre-conceptual schema: A conceptual-graph-like knowledge representation for requirements elicitation. *MICAI 2006: Advances in Artificial Intelligence*, pages 27–37.

Kaiya, H. and Saeki, M. (2005). Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In *Proceedings of the Fifth International Conference on Quality Software*, pages 223–230. IEEE.

Kaiya, H., Shimizu, Y., Yasui, H., Kaijiri, K., and Saeki, M. (2010). Enhancing Domain Knowledge for Requirements Elicitation with Web Mining. In *Proceedings of the 17th Asia Pacific Software Engineering Conference*, pages 3–12. IEEE.

Kalyanpur, A., Parsia, B., Sirin, E., Grau, B., and Hendler, J. (2006). Swoop: A 'Web' Ontology Editing Browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144–153.

Kamalrudin, M., Grundy, J., and Hosking, J. (2010). Managing Consistency Between Textual Requirements, Abstract Interactions and Essential Use Cases. In *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference*, pages 327–336. IEEE.

Kenzi, K., Soffer, P., and Hadar, I. (2010). The Role of Domain Knowledge in Requirements Elicitation: An Exploratory Study. In *Proceedings of the 5th Mediterranean Conference on Information Systems*. paper 48.

Lauenroth, K. and Pohl, K. (2008). Dynamic Consistency Checking of Domain Requirements in Product Line Engineering. In *Proceedings of the 16th IEEE International Conference on Requirements Engineering*, pages 193–202. IEEE.

Lee, J. (1991). Extending the Potts and Bruns Model for Recording Design Rationale. In *Proceedings of the 13th International Conference on Software Engineering*, pages 114–125. IEEE.

Maedche, A. and Staab, S. (2001). Ontology learning for the semantic web. *Intelligent Systems, IEEE*, 16(2):72–79.

Mostow, J. (1985). Toward Better Models of the Design Process. *AI Magazine*, 6(1):44.

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

Noy, N., Crubézy, M., Fergerson, R., Knublauch, H., Tu, S., Vendetti, J., Musen, M., et al. (2003). Protégé-2000: An Open-Source Ontology-Development and Knowledge-Acquisition Environment. In *Proceedings of the AMIA Annual Symposium*, page 953.

Noy, N., McGuinness, D., et al. (2001). Ontology development 101: A guide to creating your first ontology.

O'Madadhain, J., Fisher, D., Smyth, P., White, S., and Boey, Y. (2005). Analysis and Visualization of Network Data Using JUNG. *Journal of Statistical Software*, 10(2):1–25.

Pratt-Hartmann, I. (2003). A two-variable fragment of english. *Journal of Logic, Language and Information*, 12(1):13–45.

Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*. McGraw Hill, 6th edition.

Robinson, W. and Pawlowski, S. (1999). Managing Requirements Inconsistency With Development Goal Monitors. *IEEE Transactions on Software Engineering*, 25(6):816–835.

Ross, D. (1977). Structured Analysis (SA): A Language for Communicating Ideas. *IEEE Transactions on Software Engineering*.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented Modeling and Design*, volume 38. Prentice hall.

Schwitter, R. (2002). English as a formal specification language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 228–232. IEEE.

Schwitter, R., Kaljurand, K., Cregan, A., Dolbear, C., and Hart, G. (2008). A comparison of three controlled natural languages for owl 1.1. In *4th OWL Experiences and Directions Workshop (OWLED 2008 DC), Washington*, pages 1–2.

Siegemund, K., Thomas, E., Zhao, Y., Pan, J., and Assmann, U. (2011). Towards Ontology-driven Requirements Engineering. In *Proceedings of the 7th International Workshop on Semantic Web Enabled Software Engineering*. SWESE.

Sommerville, I. (2011). *Software Engineering*. Pearson Education Inc., 9th edition.

Sommerville, I. and Sawyer, P. (1997). *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc.

Spanoudakis, G. and Zisman, A. (2001). Inconsistency Management in Software Engineering: Survey and Open Research Issues. *Handbook of software engineering and knowledge engineering*, 1:329–380.

TONES (2008). Tones ontology repository.

Tsai, J., Weigert, T., and Jang, H. (1992). A hybrid knowledge representation as a basis of requirement specification and specification analysis. *Software Engineering, IEEE Transactions on*, 18(12):1076–1100.

Van Lamsweerde, A. (2000). Requirements Engineering in the Year 00: A Research Perspective. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 5–19. ACM.

Van Lamsweerde, A. (2001). Goal-oriented Requirements Engineering: A Guided Tour. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262. IEEE.

van Lamsweerde, A. (2003). Goal-oriented requirements engineering: From system objectives to uml models to precise software specifications. In *Proceedings of the 25th International Conference on Software Engineering*, pages 744–745. IEEE Computer Society.

Van Lamsweerde, A., Darimont, R., and Letier, E. (1998). Managing Conflicts in Goal-driven Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926.

Weston, N., Chitchyan, R., and Rashid, A. (2009). Formal Semantic Conflict Detection in Aspect-oriented Requirements. *Requirements Engineering*, 14(4):247–268.

Wiki, P. (2007).

Yu, E. (1993). Modeling Organizations for Information Systems Requirements Engineering. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 34–41. IEEE.

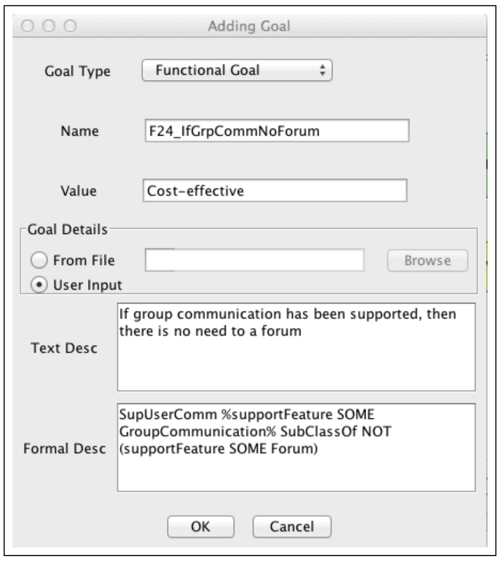
Zave, P. and Jackson, M. (1997). Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):1–30.

9 APPENDIX

9.1 ReInDetector Description

In this section, we present a scenario to demonstrate the functionality of *REInDetector*. The scenario in this demonstration is part of the traveller social networking use case which was taken in the evaluation section. The steps through this demonstration are described as followed.

1. In the first step, the requirement `F24_IfGrpCommNoForum` is added (assume that it has not been added) (cf. Figure 10).


Adding Goal

Goal Type: Functional Goal

Name: F24_IfGrpCommNoForum

Value: Cost-effective

Goal Details

From File [✓ User Input]

Text Desc: If group communication has been supported, then there is no need to a forum

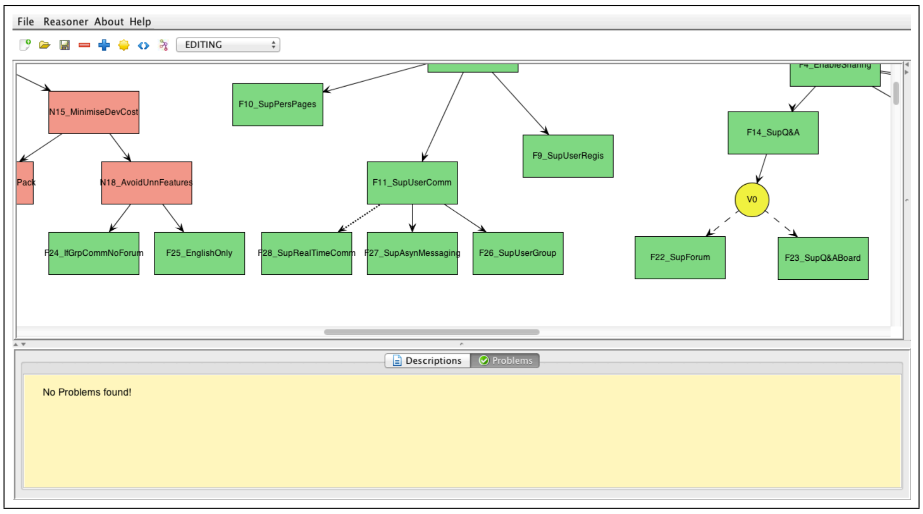
Formal Desc: SupUserComm %supportFeature SOME GroupCommunication% SubClassOf NOT (supportFeature SOME Forum)

Fig. 10 Adding new requirements.

2. In this step, we show the value of domain knowledge and semantics and how they can help in detecting requirements inconsistencies. We use
   – `F22_SupForum`: System needs to provide a forum for discussions (`System SubClassOf supportFeature SOME Forum`).

- `F26_SupUserGroup`: System supports the creation of user groups to facilitate the connection (`System SubClassOf supportFeature SOME UserGroup`).
- `F24_IfGrpCommNoForum`: If the system supports any type of group communication, then no forum is needed (`supportFeature SOME GroupCommunication NOT (SubClassOf supportFeature SOME Forum)`).

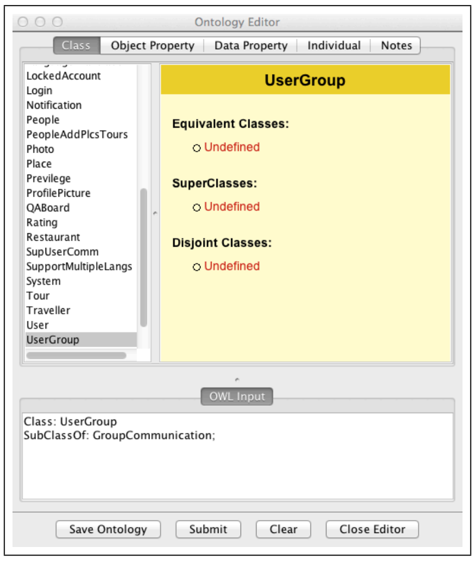
But without a proper characterization of the associated relationship between the capture concepts, no problems can manifest (cf. Figure 11). We wish to emphasize this point here, in particular, to motivate the need for proper knowledge representation in requirements engineering.



No Problems found!

Fig. 11 Requirements set with inconsistent goals.

3. Next, we demonstrate how the ontology editor can be used to define concepts, roles, instances, domain knowledge, rules, and constraints (cf. Figure 12) In particular, we show how “UserGroup” can be defined as a sub-concept of “Group-Communication”.
4. Next, we show how to use the reasoner in analyzing requirements to identify inconsistencies and how the provided explanations may help requirements engineers get more insights into the problem (cf. Figure 13).
5. In the next step, we present the requirement query support provided by *REInDetector*. We particularly make a query to analyze a partial set of requirements which doesn’t contains all requirements involved in the inconsistency. The results from the reasoner indicates that there is no inconsistency in this set of requirements (cf. Figure 14).
6. In the final step of the demonstration, we show how the identified inconsistency can be resolved. In this case, that is done by removing one of the inconsistent requirements (`F24_IfGrpCommNoForum`) (cf. Figure 15)



LockedAccount
Login
Notification
People
PeopleAddPlcsTours
Photo
Place
Previlege
ProfilePicture
QABoard
Rating
Restaurant
SupUserComm
SupportMultipleLangs
System
Tour
Traveller
User
UserGroup

UserGroup

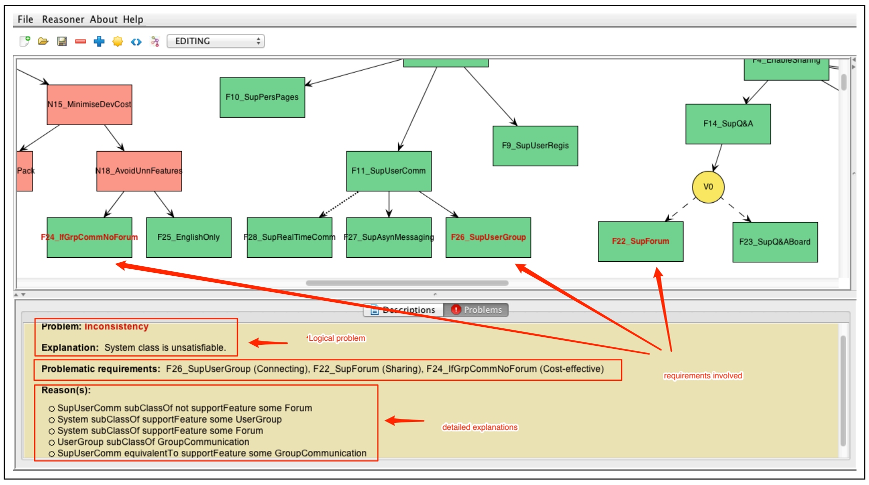
Equivalent Classes:
- Undefined

SuperClasses:
- Undefined

Disjoint Classes:
- Undefined

```
Class: UserGroup
SubClassOf: GroupCommunication;
```

Fig. 12 Defining semantics with Ontology Editor.



Problem: Inconsistency

Explanation: System class is unsatisfiable.

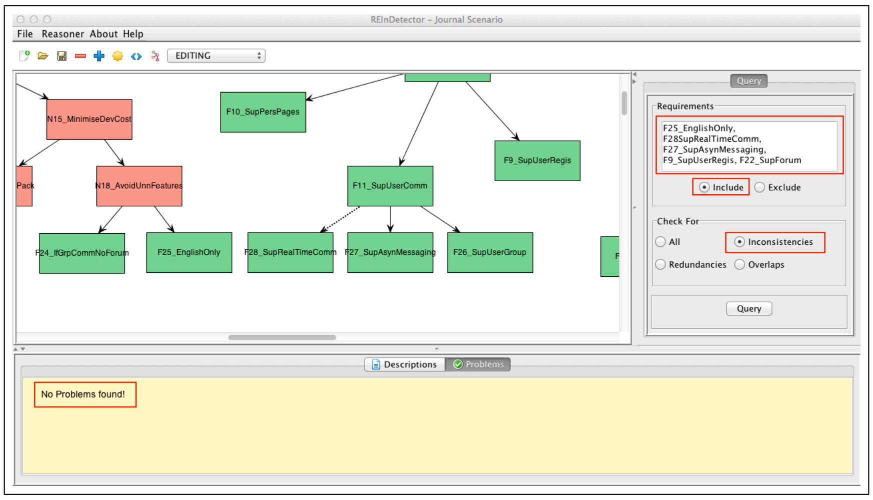
Problematic requirements: F26_SupUserGroup (Connecting), F22_SupForum (Sharing), F24_IfGrpCommNoForum (Cost-effective)

Reason(s):
- SupUserComm subClassOf not supportFeature some Forum
- System subClassOf supportFeature some UserGroup
- System subClassOf supportFeature some Forum
- UserGroup subClassOf GroupCommunication
- SupUserComm equivalentTo supportFeature some GroupCommunication

Fig. 13 A scenario of inconsistency.

9.2 Availability

REInDetector, accompanied with a demo scenario and user quick-start guide are available for download at: *http://www.ict.swin.edu.au/ personal/huannguyen/REInDetector.html*



Requirements

F25_EnglishOnly,
F28SupRealTimeComm,
F27_SupAsynMessaging,
F9_SupUserRegis, F22_SupForum

[✓ Include] Exclude

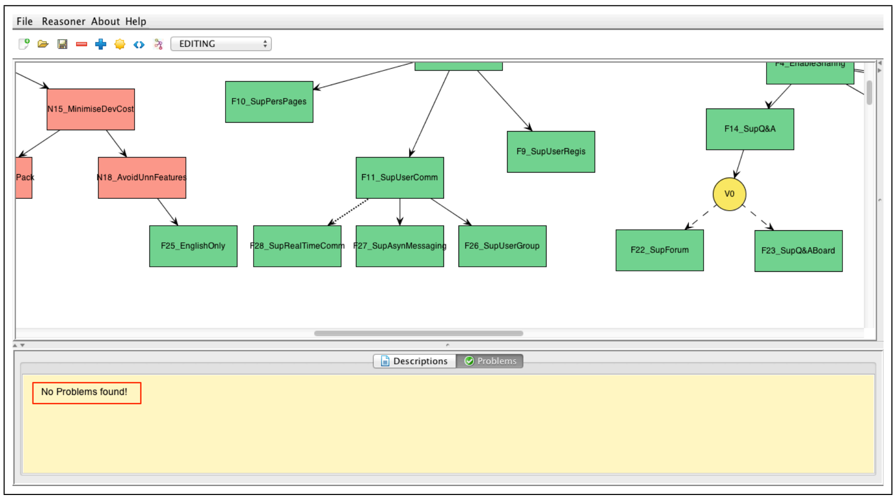
Check For

All [✓ Inconsistencies]
Redundancies Overlaps

Query

No Problems found!

Fig. 14 Requirement query service.



No Problems found!

Fig. 15 All inconsistencies resolved.