REInDetector: A Framework for Knowledge-based Requirements Engineering

Tuong Huan Nguyen, Bao Quoc Vo, Markus Lumpe, and John Grundy
Faculty of Information and Communication Technology
Swinburne University of Technology
Hawthorn, VIC 3122, Australia
{huannguyen,bvo,mlumpe,jgrundy}@swin.edu.au

ABSTRACT

Requirements engineering (RE) is a coordinated effort to allow clients, users, and software engineers to jointly formulate assumptions, constraints, and goals about a software solution. However, one of the most challenging aspects of RE is the detection of inconsistencies between requirements. To address this issue, we have developed *REInDetector*, a knowledge-based requirements engineering tool, supporting automatic detection of a range of inconsistencies. It provides facilities to elicit, structure, and manage requirements with distinguished capabilities for capturing the domain knowledge and the semantics of requirements. This permits an automatic analysis of both consistency and realizability of requirements. *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. In this paper, we describe the Description Logic used to capture requirements, the *REInDetector* tool, its support for inconsistency detection, and its efficacy as applied to several RE examples. An important feature of *REInDetector* is also its ability to generate comprehensive explanations to provide more insights into the detected inconsistencies.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*tools*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program verification*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*representation language*

General Terms

Design, Languages, Verification

Keywords

Requirements Engineering, Consistency, Description Logic

1. INTRODUCTION

Inconsistencies between requirements create major challenges for all stakeholders of a software solution and they can materialize during or between different phases of software development [9]. Conflicts arising from ill-aligned requirements typically mandate removing or modifying some of them. Yet, these changes can directly affect utility and effectiveness of the resulting system and, hence, ultimately determine a project's success. We thus require automated tool support to assist software engineers, clients, and users to make informed decisions regarding specific demands and their impact on the system design and functionality.

Requirements engineering is the process of discovering, documenting, and maintaining requirements for software solutions [14]. The key objective of these activities aims at establishing sound goals and constraints for the software system being planned and constructed. However, achieving this goal is hard. There have been numerous attempts to facilitate the elicitation of sound sets of requirements [2, 4, 6–8, 12, 15], but they lack or insufficiently support the creation and maintenance of domain knowledge and semantics of requirements that, in our view, provide the crucial elements for reliable and preferably automatic verification of, possibly hidden, requirement dependencies. Moreover, while Siegemund et al [13] propose to build an ontology to support the RE process, their focus is mainly on tracking the interactions among requirements (e.g., supporting, retracting, etc.) and the inconsistency detection is for identifying the conflicts between these interactions rather than between the requirements themselves.

Zave & Jackson [16] view requirements as *optative* concepts, whose understanding requires *domain knowledge* to help bridge between a stakeholder's intuition and what is practically implementable in a system. The domain knowledge, which also contains rules and assumptions about the system's operating environment, offers us a practical means to ensure consistency between competing objectives. Some types of requirement inconsistencies may not be detectable in the absence of such domain knowledge. Consider, for example, a scenario stipulating conflicting quality attributes for bitmaps in a graphical user interface. The requirements "*the application must support bitmaps up to 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*." Without the additional domain knowledge, we would not be able to observe the inherent conflict in the specification.

Moreover, reasoning about requirements typically mandates semantics of requirements [3]. They can be defined in terms of *instances*, *concepts* and *rôles*. Concepts are collections of modeling artifacts (*i.e.*, instances) and rôles constitute the relationships between concepts. In knowledge representation, concepts and rôles are defined abstractly in the domain-level knowledge base using specialization relationships [1]. For instance, in requirements specification, it is possible for several terms to refer to the same concept,

ASE '12, September 3-7, 2012, Essen, Germany

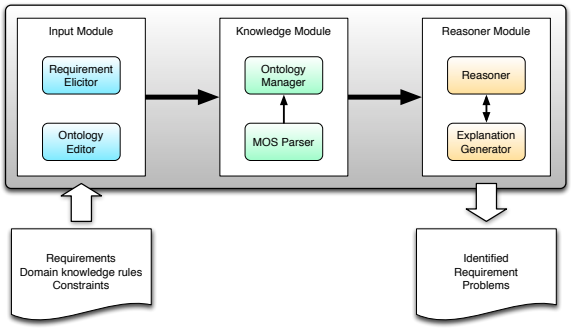

Figure 1: *REInDetector*– Overview.

such as *customer* and *client*. It is critical that these concepts are formally defined to assist the RE process. Furthermore, relationships between concepts and instances must also be captured precisely for the purpose of requirements analysis. For example, consider two requirements regarding the languages supported in an enterprise-wide system, one states that "*Chinese must be supported*" and another says that "*only major languages should be supported*." Then, 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).

As these types of reasoning are common in RE, we looked for a way to better capture it in an automated RE analysis tool. To this end, we identified *Description Logic* [1] as a suitable formalism, both expressive and usable by a wide range of RE stakeholders. In *Description Logic*, basic inference is *subsumption* (or concept inclusion). Together with domain knowledge and semantics of requirements encoded in an ontology, subsumption allows requirements engineers to unveil *implicit* consequences of *explicit* features occurring simultaneously in the requirements of a system.

2. THE FRAMEWORK

REInDetector is a Java application which provides a graphical user interface to perform all requirements **elicitation**, **management**, and **validation** tasks [15]. Our focus in *REInDetector* is on three particular objectives: a) define and maintain a knowledge base for requirements, b) provide an expressive and usable formalism for RE, and c) develop a knowledge-based RE framework to perform automated inconsistency and realizability analysis. We use *Description Logic* [1], a decidable subset of first-order logic commonly used as the formal basis of object/class-style ontologies, as the core for requirements formalization and analysis in *REInDetector*. *Description Logic* has been successfully applied to define various members of *Web Ontology Languages* (OWL) [5] and yields an attractive means to capture and maintain the domain knowledge and semantics of requirements and their corresponding relationships.

Figure 1 depicts the conceptual overview of *REInDetector*. In *REInDetector*, we support a *goal-oriented* requirements engineering approach [15] that aims at improving the manageability of requirements and allow for effective traceability of the underlining rationales of inconsistent requirements, respectively. The *Input Module* takes and manages specifications from requirements engineers, including the specific requirements, their relationships and formalizations, domain knowledge, rules, and constraints. The corresponding data is pushed into the *Knowledge Module* in which it is interpreted and stored in an ontology of *concepts*, *rôles*, and *instances*. The *Reasoner Module* provides analysis and reasoning services including inconsistency detection, requirements queries, and report generator.

We use *Manchester OWL syntax* (MOS) [10] as the requirements specification language. *Manchester OWL syntax* faithfully maps expressions in *Description Logic*. Moreover, MOS offers an easy means to the formalization of requirements (while preserving sufficient expressiveness) as it is close to natural language and much simpler than other languages such as LTL in KAOS [6] or Formal Tropos in Tropos [8]. Furthermore, by relying solely on *Description Logic* and MOS, we can utilize the off-the-shelf OWL reasoner *Pellet* [11] (with some extensions) to perform requirements queries, conflict detection and explanation services.

2.1 Input Module

The *Requirement Elicitor* enables one to capture a system's functional and non-functional requirements and shows how they are related through refinement links. Graphically, elicited goals are denoted by AND/OR graphs in which each individual goal is represented as a node annotated according to the goal's features. Each goal can be connected to other goals via refinement links (*i.e.*, edges in the graph). A refinement link not only indicates how a goal is decomposed into sub-goals, which means how a goal can be satisfied, but also reveals its parent goal, which shows the rationale behind the goal. Refinement links can include AND-connectors to indicate minimal refinements (a goal can only be satisfied if all sub-goals linked to it via AND-connectors are satisfied), OR-connectors to signify alternative refinements (a goal being refined can be satisfied by fulfilling any of its OR-connected sub-goals), and Optional-connectors to marl optional refinements (sub-goals involved in Optional-connectors are the preferred options but not strictly required for the higher-level goal to be fulfilled).

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. Both, the requirements and the ontology, need to be properly formalized in order to allow the effective analysis and reasoning on requirements. The syntax and semantics of the specification language are that of MOS, except the binary operator "**SubClassOf**" to represent requirements and the symbol "**%**" that we introduced to allow concepts to be defined within a requirement's formalization. Requirements can consist of one or more sentences, each connected by "**SubClassOf**." The expression on the right-hand side denotes the expectation or constraints for the captured concepts occurring on the left-hand side of "**SubClassOf**." In other words, "**SubClassOf**" is a primary means to define subsumption in *REInDetector*.

Figure 2 illustrates the requirement elicitation in *REInDetector*

Name	*F21_3FailsThenLocked*
Value	Security
GoalType	Functional
Refines	*F19_PreventUnAuAccess*
Refinement Link	AND
RefinedTo	none
InformalDef	If a user makes 3 failed login attempts, then their account will be locked.
FormalDef	UserWith3FailedLogins % User AND hasLogins EXACTLY 3 FailedLogin % SubClassOf User AND hasAccount SOME LockedAccount

Figure 2: Goal annotations.

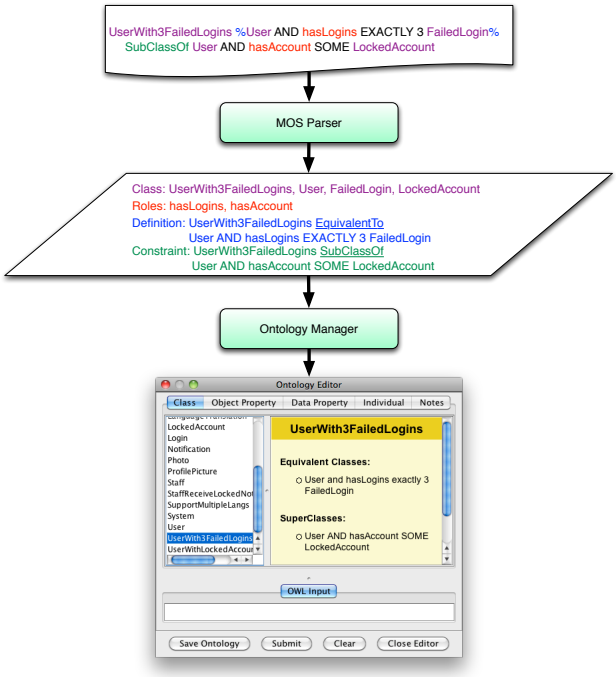

Figure 3: Ontology updated with requirements.

on a security feature that will lock an account after three failed login attempts. The annotation contains a) the details of the goal including name, value, and type, and b) the relationships to other goals including the higher-level goal it refines, the associated refinement link, and all possible depended goals (*i.e.*, "**RefinedTo**").

2.2 Knowledge Module

The *Knowledge Module* consists of two components: *Ontology Manager* and *MOS Parser*. The *Ontology Manager* handles the problem domain ontology and is the heart of *REInDetector*. We use the domain ontology to capture requirement definitions and the semantics of concepts, rôles, and instances occurring in requirements.

The *MOS Parser* converts definitions into an internal representation. It receives its input from the *Input Module* (*i.e.*, the formalizations of requirements and the domain rules and constraints) and extracts the corresponding concepts, rôles, and instances as well as their constraints and relationships. This information, in turn, becomes the input for the *Ontology Manager*, which will trigger an update of the underlying ontology.

Figure 3 illustrates in more detail for the requirement "*If a user makes 3 failed login attempts, then their account will be locked.*" The *MOS parser* takes this input and maps it to a concept called `UserWith3FailedLogins` and associates the string `%User AND hasLogins EXACTLY 3 FailedLogin%` with its definition. In addition, we obtain two rôles, `hasLogins` and `hasAccount`, and the constraint that the `User` must have an account, which has been locked after three failed login attempts: `UserWith3FailedLogins SubClassOf User AND hasAccount SOME LockedAccount`. If necessary, the *Ontology Editor* allows for further, user-specific refinements of the ontology. All modification result in either an extended ontology or an updated one.

2.3 Reasoner Module

The *Reasoner Module* is in charge of requirements validation and causal analysis. It comprises two elements: the *Reasoner* that performs the automated detection of specification issues (*i.e.*, requirements inconsistencies, redundancies, and overlaps) and the *Explanation Generator* that yields the causes (*i.e.*, explanations) for the detected problems.

The *Reasoner* is based Pellet [11], but also allows for the identification and explanation of redundancies and overlaps in requirements. These features are not supported in Pellet as of now. Moreover, we added a query facility to *Reasoner* in order to support the definition and proper handling of partial requirement specifications that can arise due to the *cyclic nature* of the requirements engineering process [14].

The *Explanation Generator* accepts "raw explanations" from the *Reasoner* and merges them with the internal requirement-constraint mappings in the ontology. As a result, we obtain a detailed report (*i.e.*, problem descriptions) as to why a specific set of requirements has failed validation. Problems can relate to either misaligned demands or logical errors within requirements. The information contained in the reports allows all stakeholders to jointly develop suitable solutions to correct erroneous requirements.

3. TOOL EVALUATION

To test the effectiveness of *REInDetector*, we run an analysis of approx. 100 requirements for a social networking system that aims at encouraging collaboration among staff in a multi-national company. We performed a controlled *failure injection*, that is, we constructed a number of non-trivial scenarios containing inconsistencies, redundancies, and overlaps in the original requirements set.

A sample run on an ill-formed set of security requirements is shown in Figure 4. The system needs to protect user account against unauthorized access. In case of tampering, the system will automatically lock an account after three failed login attempt. If this happens, the system has to issue a notice, either through SMS or email, to the user in order to the new account status. There are three possible scenarios[1]:

- `F22_IfLockedThenEmail`: If an user account has been locked, then the user will receive an account locked notification email (`UserWithLockedAccount SubClassOf User AND receiveEmail SOME AccLockedNotif`).
- `F23_IfLockedThenSMS`: If an user account has been locked, then the user will receive an account locked notification SMS (`UserWithLockedAccount SubClassOf User AND receiveSMS SOME AccLockedNotif`).
- `F24_IfEmailThenNoSMS`: If the staff member receives account locked notification via email, the this user will not receive it in SMS also (`StaffReceiveLockedNotifEmail %Staff AND receiveEmail SOME AccLockedNotif% SubClassOf Not (Staff AND receiveSMS SOME AccLockedNotif)`).

Unfortunately, these requirements employ two different terms to denote the same domain concept: "user" and "staff." Without considering the semantics and relationships between these terms, the inherent inconsistency of these requirements cannot be detected. With the use of the ontology, which captures semantics in the problem domain (*i.e.*, "staff" and "user" are equivalent), *REInDetector* can reveal the problem as shown in Figure 4.

The corresponding explanations provide a rationale governing the problem. Depending on the security and notification preferences, there are two possible solutions to rectify the problem. First,

[1]We use the prefix `F_XX` as a goal counter for requirements.

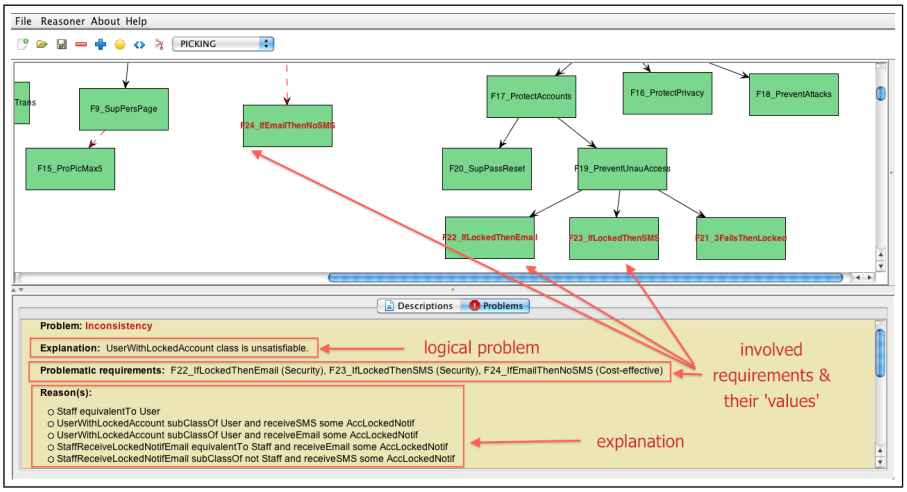

Figure 4: An identified inconsistency

if "security" is valued more than "cost effectiveness", then a resolution is to remove `F24_IfEmailThenNoSMS`, which would allow the use of both communication channels and, hence, a possibly faster notification of the user (*i.e.*, the user may not have access to email momentarily but to a smart phone or vice versa). Alternatively, we can remove either `F22_IfLockedThenEmail` or `F23_IfLockedThenSMS`, which would solve the issue also, but at the expense of "security." The user may learn of the new account state only after they has checked the system-support notification system (*i.e.*, either email or SMS). The stakeholders must make a decision here. *REInDetector* can assist them in this process.

4. CONCLUSION AND FUTURE WORK

REInDetector is a knowledge-based requirements engineering tool. It offers stakeholders an automated means to elicit, structure, and manage requirements. We use Description Logic (DL) to capture and reason about requirements as well as their constraints and relationships. DL belongs to a family of formal knowledge representation languages [1] that can effectively represent domain concepts. Artifacts (*i.e.*, requirements, concepts, rôles, and constraints) can be expressed in a quasi-natural form, a powerful tool that can assist all stakeholders in their endeavor to formulate sound sets of requirements for the software system being developed.

Requirements engineering is an iterative process. *REInDetector* can assist stakeholders in identifying the missing elements and conflicting (and possibly hidden) aspects in the requirements. Our framework encourages an incremental development approach that allows stakeholders to synchronize their assumptions and expectations. It offers a robust infrastructure to reason about requirements, even in the presence of incomplete information. Nevertheless, *REInDetector* is not able to detect the conflicts associated with the requirements that are not expressible in DL. However, this is expected due to the limited support for temporal operators in DL. Thus, a requirement such as "*When a user chooses to show their online status, the user's status button will always reflect the user's availability on the system*" can not presently be expressed in *REInDetector*. Hence, as a future extension of *REInDetector*, we seek to investigate additional constructors that would allow us to express temporal properties of the system being developed. Furthermore, we wish to explore additional abstraction mechanisms to address the varying needs of different stakeholders.

REInDetector and a user guide are available at:

http://www.ict.swin.edu.au/personal/huannguyen/REInDetector.html

5. REFERENCES

[1] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge University Press, 2nd edition, 2007.

[2] B. Boehm, P. Bose, E. Horowitz, and M. Lee. Software requirements negotiation and renegotiation aids: A theory-w based spiral approach. In *Software Engineering, 1995. ICSE 1995. 17th Int. Conference on*, pages 243–243. IEEE, 1995.

[3] T. Breaux, A. Antón, and J. Doyle. Semantic parameterization: A process for modeling domain descriptions. *ACM TOSEM*, 18(2):5, 2008.

[4] K. Chung. Representing and using non-functional requirements: a process-oriented approach. *IEEE Trans. Software Engineering*, 18(6):483–497, 1993.

[5] O. Corcho and A. Gómez-Pérez. A roadmap to ontology specification languages. In *EKAW 2000*, pages 80–96, 2000.

[6] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[7] A. Egyed and P. Grunbacher. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, 21(6):50–58, 2004.

[8] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in tropos. *Requirements Engineering*, 9(2):132–150, 2004.

[9] P. Henderson. Why large it projects fail. *ACM Trans. Program. Lang. Syst*, 15(5):795–825, 2006.

[10] M. Horridge, N. Drummond, J. Goodwin, A. Rector, R. Stevens, and H. Wang. The manchester owl syntax. *OWL: Experiences and Directions*, pages 10–11, 2006.

[11] Pellet – OWL 2 reasoner, http://clarkparsia.com/pellet/.

[12] W. Robinson and S. Pawlowski. Managing requirements inconsistency with development goal monitors. *IEEE Trans. on Software Engineering*, 25(6):816–835, 1999.

[13] K. Siegemund, E. Thomas, Y. Zhao, J. Pan, and U. Assmann. Towards ontology-driven requirements engineering.

[14] I. Sommerville. *Software Engineering*. Pearson Education Inc., 9th edition, 2011.

[15] A. Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings of 5th International Symposium on Requirements Engineering*, pages 249–262, 2001.

[16] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM TOSEM*, 6(1):1–30, 1997.