Software Qual J (2016) 24:635–673
DOI 10.1007/s11219-015-9281-7

Ontology-based automated support for goal–use case model analysis

Tuong Huan Nguyen[1] · **John C. Grundy**[1] · **Mohamed Almorsy**[1]

Published online: 9 June 2015

Abstract Combining goal-oriented and use case modeling has been proven to be an effective method in requirements elicitation and elaboration. To ensure the quality of such modeled artifacts, a detailed model analysis needs to be performed. However, current requirements engineering approaches generally lack reliable support for automated analysis of consistency, correctness and completeness (3Cs problems) between and within goal models and use case models. In this paper, we present a goal–use case integration framework with tool support to automatically identify such 3Cs problems. Our new framework relies on the use of ontologies of domain knowledge and semantics and our goal–use case integration meta-model. Moreover, functional grammar is employed to enable the semiautomated transformation of natural language specifications into Manchester OWL Syntax for automated reasoning. The evaluation of our tool support shows that for representative example requirements, our approach achieves over 85 % soundness and completeness rates and detects more problems than the benchmark applications.

Keywords Goal-oriented requirements engineering · Ontology-based analysis · Requirements incorrectness · Incompleteness and inconsistency detection

✉ Tuong Huan Nguyen
huannguyen@swin.edu.au

John C. Grundy
jgrundy@swin.edu.au

Mohamed Almorsy
malmorsy@swin.edu.au

[1] Faculty of Science, Engineering and Technology, Swinburne University of Technology, Melbourne, VIC, Australia

1 Introduction

Requirements engineering (RE) is an iterative process of eliciting, structuring, specifying, analyzing and managing software requirements (Sommerville 2011). Within this process, goal-oriented RE (Anton 1996; Van Lamsweerde 2001) and use case-based modeling (Glinz 2000; Sutcliffe 2003) are two key approaches. However, both of them have limitations including: Many goals do not readily exist in practice, and the discovery and refinement of them can be challenging (Rolland et al. 1998); it is often not easy for stakeholders to express goals at the required level of abstraction (Van Lamsweerde and Willemet 1998); and use cases are fragmented, in the sense that different single uses of the system are described, while the underlining rationale and relationships between them are not always known.

A number of attempts have been made to combine these concepts to leverage the benefits while overcoming the disadvantages of either approach (Anton et al. 2001; Rolland et al. 1998; Kim et al. 2006; Van Lamsweerde and Willemet 1998). However, these approaches generally lack support for automated analysis of goals and use cases in the same model for inconsistency, incompleteness and incorrectness. Although various requirements analysis techniques exist, ranging from formal methods (Van Lamsweerde et al. 1998; Lee et al. 1998), natural language (Gervasi and Zowghi 2005; Lami et al. 2004) to ontology-based techniques (Nguyen et al. 2014; Kaiya and Saeki 2005), they suffer from one or more of the following problems:

- Lack of comprehensive support for the analysis of goal–use case models: Most analysis techniques are proposed for tackling problems in goal models [i.e., KAOS (Van Lamsweerde 2001), Tropos (Fuxman et al. 2004)], use case models [i.e., (Lee et al. 1998)] or general requirements [i.e., (Gervasi and Zowghi 2005; Lami et al. 2004)]. They thus are not able to validate the alignment between goals and use cases. In addition, many of the above questions are not handled in these approaches.
- Require complicated requirements modeling languages: Many formal approaches to automated requirements analysis require the use of complex modeling languages [i.e., KAOS (Van Lamsweerde 2001), Tropos (Fuxman et al. 2004)]. This creates a *pragmatic barrier,* which limits their applicability in practice (Dwyer et al. 1999).
- Lack of support for identifying semantic-related problems: Identifying semantic problems in goal–use case models requires the meaning of vocabularies and knowledge in the requirements domain to be captured and used. A number of ontology-based approaches have been proposed to tackle semantic requirements defects (Nguyen et al. 2014; Kaiya and Saeki 2005); however, these techniques are neither adequately mature nor suited to goal–use case model analysis.

In this paper, we propose a novel Goal and Use case Integration Framework (GUI-F) to complement existing approaches with automated support for the detection and resolution of incompleteness, incorrectness and inconsistency in natural language goal and use case specifications. GUI-F relies on Functional Grammar (Dik 1989) to enable the semiautomated transformation of natural language specifications into Manchester OWL Syntax (Horridge et al. 2006) for automated reasoning. In addition, GUI-F incorporates our goal–use case integration meta-model and domain ontologies to allow both syntactic and semantics analysis. Moreover, GUI-F allows the automated generation of comprehensive explanations for detected problems and is able to suggest repairing alternatives.

The rest of the paper is organized as follows. In Sect. 2, we describe the motivation of this work. Section 3 presents some background knowledge. Section 4 provides the

overview of our approach. Section 5 presents the description of our goal–use case integration meta-model. Discussion about GUI-F's main identifiable 3Cs problems and associated detection and resolution techniques can be found in Sect. 6. Section 7 presents usage examples. Evaluation results are provided in Sect. 8 followed by a discussion on the advantages and challenges of GUI-F in Sect. 9. We summarize the existing key techniques in Sect. 10. Section 11 summarizes our contribution.

2 Motivation

Consider a group of requirements engineers who are distilling the requirements for a Traveler Social Networking system, intended to improve the quality of travel planning (of travelers). They have obtained a set of goals and use cases and want to combine them into a single model to acquire more comprehensive requirements. Figure 1 depicts a partial goal–use case model they have developed (a sketch notation for goals and use cases is used. It is not bound to any particular language). The top artifact denotes the objective of the system (i.e., improve the quality of travel planning). It is refined into a number of lower level goals. Such goals are then operationalized by use cases. They face, however, a number of problems about the incompleteness, incorrectness and inconsistency of these artifacts. In the following subsections, we discuss these problems in details. For readability reason, we have placed labels underneath each artifact to indicate in which example it is referred to (i.e., Ex1 means example 1).

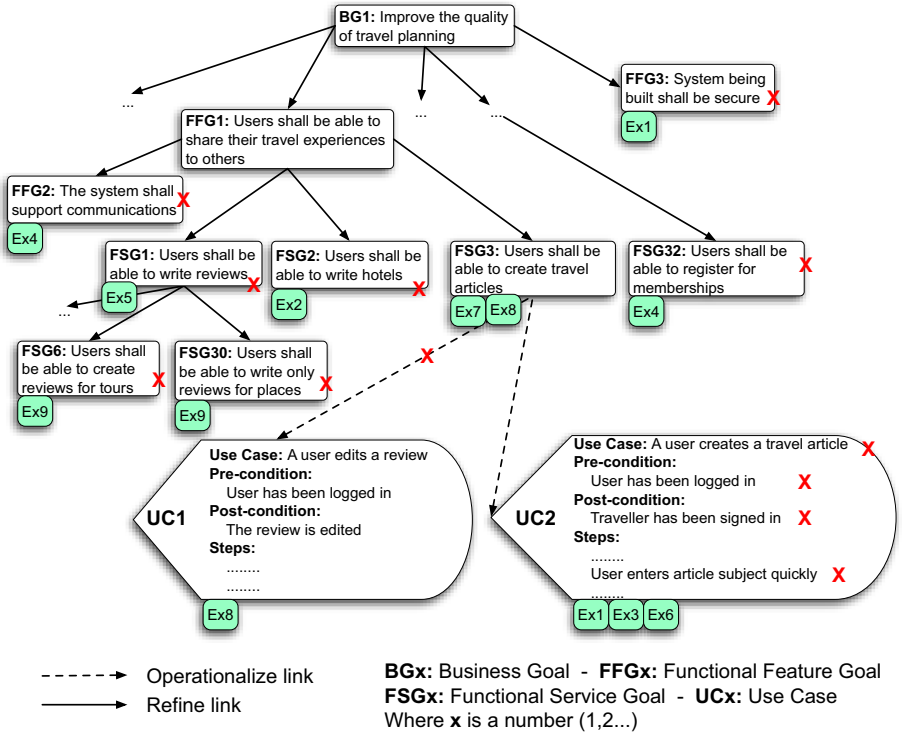


Fig. 1 Partial goal–use case model for the traveler social network system

2.1 Incorrectness

In our work, correctness refers to the correspondence between artifact specifications and the system's needs and constraints. In other words, it is concerned with the soundness of an artifact specification. In the following examples, the obtained goals and use cases may be incorrect due to their unsound specifications or relationships.

Example 1 Consider the functional goal FFG3 "The system being built shall be secure" and UC2's use case step "User enters article subject quickly," These specifications are malformed for their types. The former should describe a system's functionality rather than quality, while the later should not state how a user achieves a task.

Example 2 Consider the goal FSG2 defined as "Users shall be able to write hotels." It is not correct semantically because *hotels* cannot be *written*.

Example 3 Consider use case UC2 which has the precondition "user has been logged in" and the post-condition "traveler has been signed in." Such use case specification is invalid as its post-condition is identical to its precondition, given that *user* is equivalent to *traveler* and *logged in* is identical to *signed in* in this domain.

2.2 Incompleteness

Completeness is the quality that indicates whether all the needs and constraints for the system have been identified. In the following examples, incompleteness may include missing artifacts that need to be elicited and modeled, missing parts in artifact specifications or missing relationships between modeled artifacts.

Example 4 Consider two goals: FSG32 "Users shall be able to register for memberships" and FFG2 "System shall support communication." The former requires an operationalizing use case to describe the needed system–user interaction for membership registration. The later needs to be further refined into a specific goal since it is necessary to identify more specific goals that specify what types of communication to be supported by the system. However, these needed artifacts have not been specified.

Example 5 Consider the goal FSG1 "Users shall be able to write reviews." There is a missing goal about "editing reviews" given the general domain knowledge that any *content writing* activity is usually accompanied with a *content editing* activity.

Example 6 Consider the use case UC2 ("A user creates a travel article"). Assume that in this domain a *banned user* is not allowed to create a review. Thus, if no precondition or extension defined to handle this case, there is possibly incompleteness.

Example 7 Consider goal FSG3 "Users shall be able to create travel articles" and use case UC2 "User creates a travel article." Assume they are not connected, then that is incompleteness since UC2 describes the step to realize the goal specified by FSG3.

2.3 Inconsistency

Consistency refers to the requirements model quality, indicating that no two or more specifications contradict each other. In the following examples, inconsistency can be the

discrepancies between a goal and its associated use case's specification, illogical relationships between artifacts or conflicting artifact specifications.

Example 8 The goal FSG3 "Users shall be able to create travel articles" is operationalized by the use case UC1, which has the description of "A user edits a review." This is considered an inconsistency because the use case needs to describe the steps to achieve the goal it operationalizes.

Example 9 Consider two goals, FSG6 "Users shall be able to create reviews for tours" and FSG30 "Users shall be able to write only reviews for places." They are inconsistent given "write reviews" and "create reviews" are equivalent activities, while *tour* and *place* are disjoint concepts in the domain.

Our motivating examples suggest the following solution for automating the detection of inconsistency, incompleteness and incorrectness in goal–use case models:

1. Certain syntactic and semantic definitions must be given to different types of artifacts and their relationships to regulate how each artifact should be specified and connected to another. The artifacts should cover the common concepts in goal and use case-based RE and various relationships between the concepts.
2. Artifact specification should be structured in a way so that their semantics can be captured and analyzed effectively by computers.
3. The ability to track the semantics of vocabularies and terms used in artifact specifications is required to enable the identification of semantic-related problems.
4. The solution should allow requirements engineers to work with natural language specifications as they are predominantly used to specify requirements in practice.

3 Background

In this section, we provide some background information on functional grammar, ontology and how we employ these concepts in our approach.

3.1 Functional grammar

Functional grammar (FG) is a general theory concerning the grammatical organization of natural languages (Dik 1989). In GUI-F, FG is used to parameterize artifact textual specifications into different parts called semantic functions. Such parameterization provides a standard way to interpret the semantic role of each group of words in a statement and thus offers means to analyze the semantics of specifications.

Figure 2 shows the key components of a statement as used in our work (i.e., goals, use case steps/conditions). A statement consists of four predicates, each denoting a number of semantic functions. For instance, *nuclear predicate* contains elements describing which action is conducted (*verb*), by whom (*agent*), or on what target (*object*). *Core predicate* enriches *nuclear predicate* with details about the *beneficiary* or how an activity is performed (*manner*). Each semantic function is described by a *term*. For instance, *verbal terms* describe activities and thus are used to specify statements; *nominal terms* denote entities and are used to specify the values of most semantic functions. For example, the nominal term *Head*(*Review*) + *Quantifier*(*Quanti-ty*(*2*) + *Comparative_Operator*(*More_*

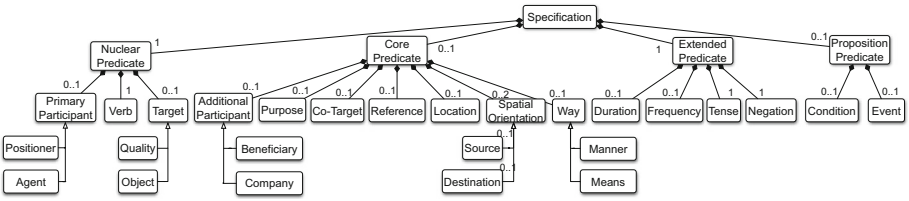


Fig. 2 The structure of a statement

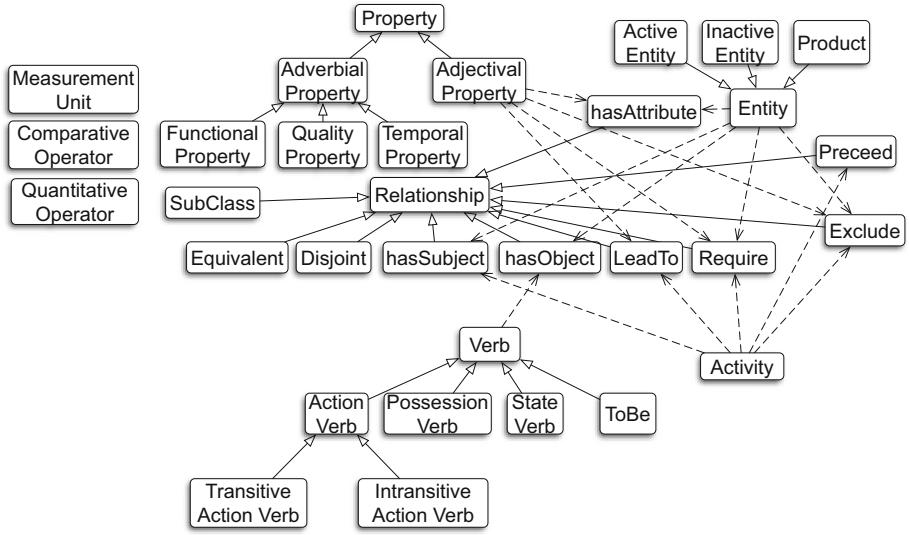


Fig. 3 Ontology structure

Than_Or_Equal)) + *Quality*(*Attribute*(*High_ quality*)) describes the phrase "2 or more high quality reviews."[1]

3.2 Ontology

3.2.1 Ontology definition and representation

Ontology is a repository of domain concepts and their associated relationships. In our work, ontology is intended to capture semantics of terms. Using FG, a statement can be semiautomatically parameterized into a set of atomic ontological items whose semantics are known. This provides the basis upon which artifacts can be analyzed. Figure 3 depicts the ontology meta-model. Some main concepts and relationships are as follows:

[1] For a detailed account of FG, the interested reader is referred to (Nguyen 2014).

- *Verb* refers to verbs in the domain that describe actions (i.e., display, show, create), possession (i.e., have, contain) or statuses (i.e., to be, arrive, come). Action verbs are further classified as *transitive action verbs* and *intransitive action verbs*.
- *Entities* are core elements in the domain. *Product entities* refer to the software system and its components (e.g., *system*, *product*). *Active entities* are the rest of the entities that can perform an action (e.g., *user*, *librarian*). *Inactive entities* are not allowed to perform actions; they are objects of actions (e.g., *book*, *password*).
- *Measurement unit* refers to measurement units in the domain. They can be classified further (e.g., data storage units like *MB*, or time units like *MHz*).
- *Property* includes adjectival properties (e.g., *high*, *low*) and adverbial properties. Adverbial properties are classified into functional properties (e.g., *automatically*, *manually*) and qualitative properties (e.g., *quickly*, *safely*).
- *Equivalent, Subclass and Disjoint,* respectively, specify analogous, refinement and non-overlapping relationships between concepts.
- *Require/Exclude* specifies that a concept requires or contradict to another (i.e., "*banned users* are not allowed to *create reviews"* is captured by an "exclude" relationship between the "banned user" entity and the "create review" activity).

.

3.2.2 How to build and maintain ontology?

In this section, we describe our steps to create ontologies. We adapted Noy et al. ontology building guidelines (Noy and McGuinness 2001). Protégé[2] is our choice for ontology development tool due to its comprehensive features. The examples used in this section come from our attempt to build ontology for the traveler social networking domain (used in our validation).

Step 1: Determine the domain and scope of ontology It is important to identify in which domain we are going to build ontology. A domain often has a number of sub-domains. For instance, the sub-domains of the traveler social networking domain include: travel, online community and web application. Moreover, the ontology's scope needs to be determined. In the context of our work, ontology should contain the concepts and their relationships in the relevant domain. Our ontology meta-model (cf. Fig. 3) suggests what to look for to build ontology. For instance, they include:

- What the entities and properties used to describe the domain are (i.e., *traveller*, *booking*, *safe*, *expensive*).
- What activities to be carried out (i.e., *create review*s, *book hotel*s).
- What the characteristics of such activities are. For instance, who book hotels? Who write reviews? Who is not allowed to write reviews?

Step 2: Consider reusing existing ontologies It is important to consider existing ontologies and check whether they can be reused. In this step, the sub-domains identified in step 1 become extremely important. If no existing ontology were found for the main domain, looking into the sub-domains would be helpful in obtaining existing relevant ontologies. For instance, since no ontology was found for the *traveller social networking* domain, we then obtained relevant ontologies in the domains of *travel* and *online*

[2] http://protege.stanford.edu/.

community. Some ontology repositories such as Protégé Ontology Library[3] and Swoogle Semantic Web Search Engine[4] offer pre-built ontologies in different domains. Once relevant ontologies are obtained, the following steps are performed.

Step 2.1: Study the ontologies' contents and structures It is critical to gain the understanding about what each ontology contains and how it is specified (i.e., its naming convention and class hierarchy). Such information is very important when merging these ontologies into one, especially if they are overlapped

Step 2.2: Merge the ontologies into a single ontology We use PROMPT (Noy and Musen 2003), a plugin for Protégé, to merge multiple ontologies. PROMPT provides a semiautomated and interactive support for (1) identifying identical or linguistically similar concepts in two ontologies, (2) suggesting merging actions and (3) identifying and resolving conflicts in the resulting ontology. PROMPT reportedly achieved over 90 % precision and recall rates for its suggestion quality

Step 2.3: Convert the resulting ontology into our format We need to ensure the ontology is in Manchester OWL Syntax (the format our framework accepts). This can be done automatically by Protégé. Moreover, it is needed to make sure the ontology is specified according to our ontology meta-model. For instance, we use our notion of *activity* instead of object properties to describe the relationships between classes to better capture the relationships between activities in the domain

Step 3: Semiautomatically build an initial ontology In case no existing ontology can be obtained, or if existing ontologies are found incomplete or unsuitable for our needs, we can start exploring ontology components from our obtained domain descriptions. We use Text2Onto (Cimiano and Völker 2005), an ontology learning tool, to build an initial ontology. Text2Onto provides a semiautomated support for extracting ontologies with concepts and relationships (i.e., subclass, instances) from textual domain descriptions. It thus helps us save time and effort to build an initial version of the ontology. Text2Onto, however, has a number of limitations. First, it identifies all noun phrases in a sentence as concepts that may create false-positives. Therefore, after the initial ontology is obtained, it is necessary to manually verify the validity of the identified concepts and relationships. Second, it does not support the identifications of quality properties (i.e., good, expensive), activities and a number of relationships (i.e., require, leadTo). Thus, additional steps need to be taken to further complete the ontology. If an existing ontology were used, it would be merged into the initial ontology

Step 4: Identify and classify concepts in the ontology This step focuses on identifying additional concepts in the domain that have not been included in the initial ontology. We also classify all concepts according to our ontology's meta-model. For example, *traveller* should be classified as an active entity, *hotel* as an inactive entity, "reserve room" as an activity or *secure* as an adjectival property. We have developed our classification of around 300 commonly used verbs and their relationships (i.e., *equivalent*, *disjoint*).[5] Therefore, in this step, verbs do not need to be identified and classified unless they do not exist in our collection

Step 5: Identify relationships between concepts In this step, we identify additional semantic relationships between the concepts, for instance, *equivalent*, *disjoint*, *require*,

[3] http://protegewiki.stanford.edu/.

[4] http://www.swoogle.umbc.edu.

[5] The verb collection can be found at http://goo.gl/gCUofM.

exclude, *leadTo* (between activities) and *hasAttribute*. In textual domain descriptions, these relationships are usually recognized by a number of grammatical structures. For example, a *require* relationship between the activities "register account" and "post comment" can be identified from the sentence "a user must register for an account before he can post comments in the system." In this example, the structure "must do something before doing something" implies a *require* relationship.

Ontology building is a time- and effort-consuming task. It is necessarily an iterative process. For instance, in case step 5 is already completed, if new domain documentations are available, we would need to go back to step 3 to identify and classify additional terms. A sample ontology that we have created for the domain of traveler social networking can be found at http://goo.gl/gCUofM.

3.2.3 How to evaluate ontology's quality?

It is critical to ensure ontology is error free before it can be used. Table 1 presents a categorization of ontology anomalies in the context of our work. These categories were developed based on a collection of general ontology pitfalls identified and investigated by researchers and practitioners (Poveda-Villalón et al. 2012). We identified the list of applicable anomalies and extended it with possible issues in ontology specified in our format. The anomalies are classified into three categories: incorrectness, incompleteness and inconsistency. The complete descriptions of these anomalies can be found at http://goo.gl/gCUofM.

In GUI-F, we use OOPS! (OntOlogy Pitfall Scanner),[6] a web-based ontology validator, to identify these anomalies. OOPS! is able to automatically identify A2, A3, A8–11 and A14 (cf. Table 1). Our ontology editor can automatically detect A1 and A4. A5, A6, A7, A12 and A13 are semantic problems. For instance, A5 is concerned with the semantic validity of a relationship between two classes (i.e., does activity A **really** require activity B?). A12 and A13 are about the possibility of a class or a relationship in the domain of interest has not been identified. We partially support the identification of these problems. For instance, our ontology editor detects the cases when not all predefined classes (i.e., active entity, inactive entity, adjectival property) have a subclass. In addition, it identifies classes that are not connected to any pre-defined classes or does not have all of its predefined relevant relationships specified (i.e., an activity that does not have a require relationship with any other activity). Such detection helps ontology engineers enhance the ontology completeness.

4 Our approach

We have devised GUI-F (goal and use case integration framework) and an associated tool support to provide the automated detection and resolution of inconsistency, incompleteness and incorrectness in goal and use case models.

Our framework encompasses a two-layered goal and use case integration meta-model, in which an *artifact layer* hosts the modeled artifacts (i.e., goals, use cases and their relationships), and a *specification layer* captures the rules governing artifact composition. This meta-model provides the foundation for identifying syntactic problems (i.e., a goal is

[6] http://oops.linkeddata.es/.

Table 1 Categories of ontology anomalies

Incorrectness
A1: creating unclassified classes
A2: merging different concepts into the same class
A3: missing annotations
A4: syntactically invalid relationships
A5: semantically invalid relationships
A6: specifying a hierarchy exceedingly
A7: specifying the subject of an activity and object of a verb exceedingly
A8: wrapping intersect ion and union
A9: misusing ontology annotations
A10: using a miscellaneous class
A11: using different naming criteria
Incompleteness
A12: missing classes
A13: missing relationships
Inconsistency
A14: ontology is logically inconsistent

not specified correctly according to its type). The formation of artifact specifications is inspired by functional grammar, in which a statement is described by a verb and a number of parameters, where each parameter has its own semantic function (as discussed in Sect. 3). For instance, the parameterization of the goal "Users shall be able to create reviews easily" is "Agent(User) + Verb(Create) + Object(reviews) + Manner(easily)." Such a parameterized expression is called *structured specification* in GUI-F. In order to assign specification semantics, we equip our framework with domain ontologies, integrated with the functional grammar-based parameterization. That provides us with means to automatically identify 3Cs problems within a goal–use case model. In order to assist requirements engineers to specify artifacts in our framework, we provide a set of boilerplates developed based on artifact composition rules according to the meta-model's specification layer. These boilerplates define the specification templates for writing goals and use cases.

Figure 4 shows the modeling and analysis processes in GUI-F. Both processes are governed by the underlining goal–use case integration meta-model (discussed in Sect. 5). Requirements engineers start from textual requirements documents and extract goals and use case specifications from such documents. These specifications are entered into GUI-F using a database of boilerplates (1.1). The entered textual specifications are then automatically transformed (2) into structured specifications that will then be used in the analysis step. The relationships between artifacts (i.e., a goal refines another goal, or a use case operationalizes a goal) also need to be acquired from the requirements documents (1.2). After having modeled both specifications and relationships, requirements engineers obtain a goal–use case model (a goal–use case model contains goal and use case specifications and their relationships) (3). Different requirements elicitation and elaboration techniques (e.g., derive use cases from goals and vice versa) can be applied in this phase to complete the model.

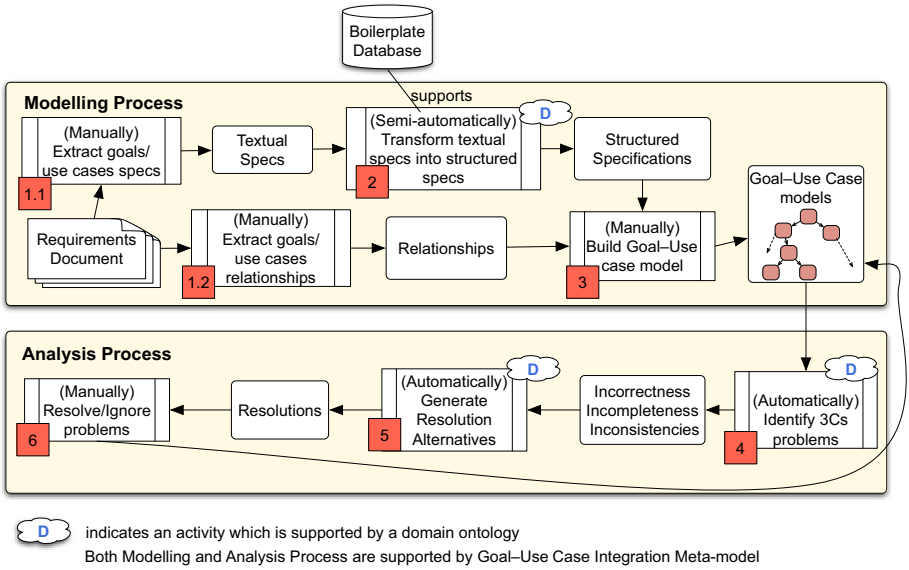


Fig. 4 The GUI-F usage process

The analysis process involves the automated detection of incorrectness, incompleteness and inconsistency (4) and generation of resolution alternatives (5). In addition, for each identified problem, GUI-F provides comprehensive explanations to help requirements engineers to get more insight into the problems and make appropriate responses. Requirements engineers can choose to ignore the problems or select one of the resolution alternatives or use their own repairing strategies (6).

5 Goal–use case integration meta-model

In GUI-F, this meta-model is the underlying foundation for the entire modeling and analysis processes. It contains two layers as follows.

5.1 The artifact layer

As determining abstraction levels of goals is usually a complicated task (Van Lamsweerde and Willemet 1998), defining different levels of abstraction and the relationships between goals is critical in the goal refinement and elaboration process. Figure 5 illustrates the *artifact layer*, which defines:

- *Business goals* (BG) that describe the business objectives of the software system (e.g., "Improve quality of travel planning").
- *Functional feature goals* (FFG) that list features the system should support in order to achieve business goals. FFGs should not offer details as to what functions are needed to support a feature (e.g., "System shall support communication").
- *Functional service goals* (FSG) provide the details of how a feature is achieved. A FSG describes what function a user or the system can perform. The difference between a

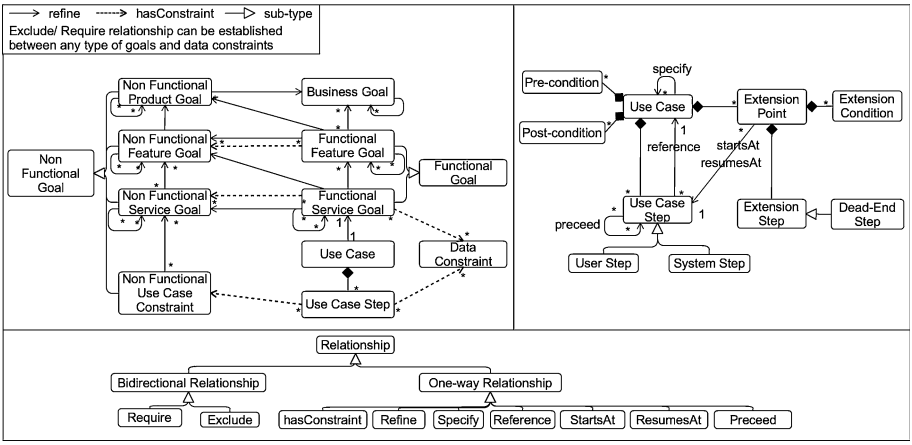


Fig. 5 Artifact layer

FSG and a FFG is that, a FSG is detailed enough to form a testable unit and is operationalized by a use case (e.g., "Users shall be able to create reviews").

- *Non-functional product goals* (NPG) describe quality constraints on the entire product (e.g., "The system being built shall be secure").
- *Non-functional feature goals* (NFG) name quality constraints of a particular feature (i.e., specified in a FFG). A NFG does not contain detailed information about how such constraint can be met (e.g., "Users shall be able to share experience easily").
- *Non-functional service goals* (NSG) that identify quality constraints on a particular service. For example, "The article creation process is familiar to typical Internet users" is a NSG restricting the FSG "Users shall be able to create travel articles."

A use case specification includes pre/post-conditions, steps and extensions. They operationalize functional service goals. A *non-functional use case constraint* describes a quality constraint at use case level (e.g., "System validates a user's identity within 2 s"). A *data constraint* captures a data requirement for an entity mentioned in a functional service goal or use case (e.g., "A review contains a subject, a rating, and a co**mme**nt)." We also allow for common relationships between artifacts such as *refine*, *require* and *exclude*. Figure 6 provides a partial goal–use case model with these types of artifacts highlighted.

5.2 Specification layer

This layer imposes rules on how each artifact should be specified based on FG. In other words, it provides guidelines for writing artifacts as to which semantic functions should and should not be used for a certain artifact. For example, as business goals are usually high-level strategic statements, *condition* or *duration* should not be specified, while other parameters (i.e., *beneficiary*, *destination*) are permitted (cf. Fig. 7). Based on that, the specification layer can assist to detect some types of 3Cs problems. Figure 7 shows compulsory and optional semantic functions of a business goal's specification. For instance, the nuclear predicate's possible semantic functions are *verb*, *object*, *location*, *source* and *destination* in which *verb* and *object* are compulsory components. Consider, for example, a valid business goal "Assist travellers to plan their travels," parameterized as

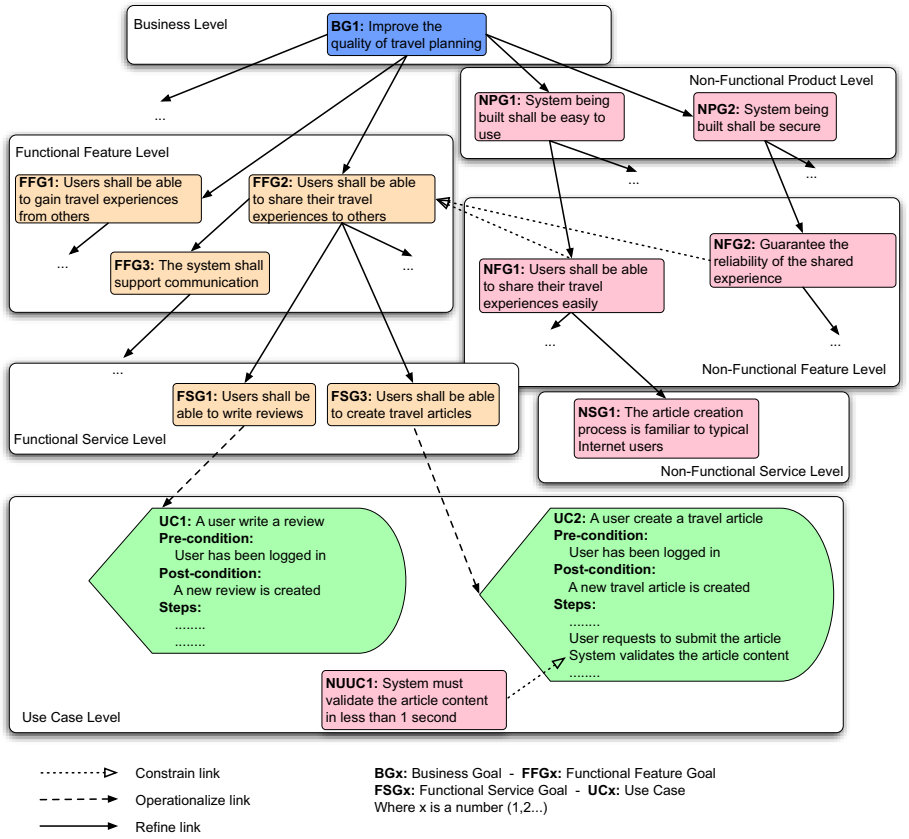


Fig. 6 Sample goal–use case model

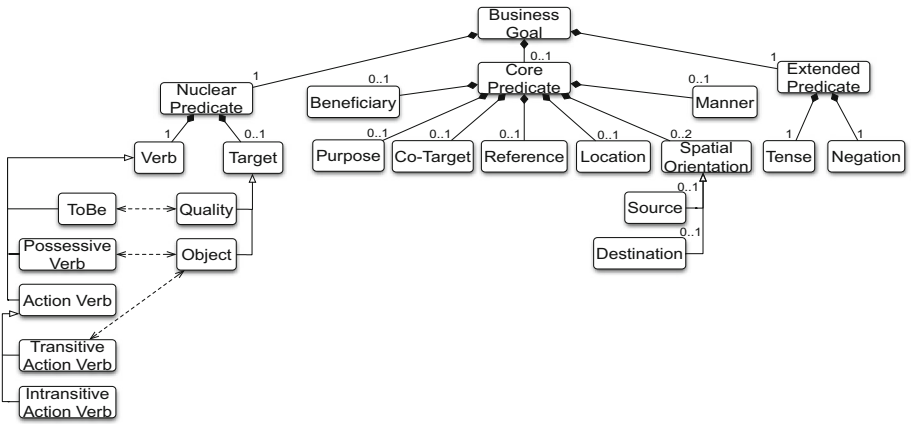


Fig. 7 Specification rule for business goals

"Verb(assist) + Object(Agent(traveler) + Verb(plan) + Object (travel)) + Tense(Present) + Negation(false)." If a *condition* were specified (i.e., "If the demand is high"), the specification would become invalid because a *condition* semantic function is not permitted here. The specification models for other artifacts can be found in (Nguyen 2014).

6 3Cs Problem detection and resolution

6.1 Problem detection

In GUI-F, incorrectness, incompleteness and inconsistency are classified as syntactic or semantic problems. The meta-model is used to detect syntactic problems, while ontologies are used for semantic problems. Both are supported by structured specifications in FG. Table 2 presents the main problems GUI-F deals with and techniques used to detect them. These techniques are described in the following subsections.

T1: Meta-model matching This technique is used to identify syntactical incompleteness and incorrectness. The key idea is to use our meta-model to validate a specification or relationship. Specifically, the artifact layer is used to validate artifacts, relationships and identify missing artifacts. In example E2, a functional service goal is missing because according to the artifact layer, functional feature goals are on high levels and must be further refined by service goals. The specification layer is used with structured descriptions (by FG) to identify missing or invalid parts in a specification. In example E1, the structured description of the goal is "Agent(Head(User)) + Verb (Create)." An incompleteness is identified because according to the specification layer, an *Object* semantic function must be defined in a functional goal's specification. Table 3 presents the rules used by this technique to identify problems.

T2: Relationship checking/inferring This technique is used to identify semantic incompleteness or incorrectness. Its key idea is to extract an activity from a structured specification and use ontologies to identify its relevant relationships. In example E5, the exception from the use case is detected by first extract the activity in the use case's description (i.e., *Verb*(*Create*) + *Object*(*Head*(*Review*))), then transform it into a MOS description (i.e., *hasVerb SOME Create AND hasObject SOME Review*). The next step is to check the ontology for an "exclude" relationship between that *activity* and an *entity* (i.e., "create review" *excludes* "banned user"). If the entity is a subclass of the *agent* in the use case's description (i.e., *User*) and if there is no precondition or extension handling this exception, then there is incompleteness. Note that it does not need to be an explicit relationship between "create review" and "banned user" in the ontology. Such relationship can be inferred in GUI-F based on the semantics of concepts in the ontology. Table 4 presents our algorithm to detect missing relationships. Algorithms for other problems solvable by T2 follow the same concepts.

T3: Parameter matching This technique is used to detect the equivalence, inconsistency or overlap between two specifications. The key idea is to compare the corresponding parameters in both structured descriptions. In E15, the goals are "Agent (User) + Verb(Write) + Object(Head(Review) + Object (Tour))" and "Agent(Traveller) + Verb(Create) + Object(Head(Review) + Object(QuantitativeOperator(Only) + Head(Place)))." The inconsistency is detected as the parameter-to-parameter semantic

Table 2 Main identifiable problems and associated detection techniques (tnq.) in GUI-F

Problem sub-type	Examples	Tnq
Incompleteness		
Syntactic		
Incomplete artifact spec	E1: a goal "users shall be able to create" missing an object of action "create"	T1
Missing artifact	E2: a functional feature goal is not refined by any functional service goals	
	E3: no security constraint has been specified (in case the system being developed requires security features)	
Missing use case component	E4: A use case that does not have one or more required components, i.e., precondition, steps	
Semantic		
Missing exception handling	E5: in the "user creates a review" use case, the exception case when a *user* is *banned* (not allowed to create reviews) has not been handled	T2
Missing required/sub/parent goals	E6: goal "users shall be able to create reviews" is specified while the goal "users shall be able to edit reviews" is not (assume the general knowledge that if a content can be created, it can then be edited/updated)	
Missing relationships between artifacts	E7: a "require" relationship is missing between goals G1 "users shall be able to edit reviews" and G2 "users shall be able to create reviews" (same reason as above)	
	E8: A use case with description "a traveller edit a review" is not linked to G1	
Incorrectness		
Syntactic		
Syntactic incorrect artifact spec	E9: a use case step "User enters article subject quickly" is ill formed, as the quality attribute ("quickly") should not be used in a use case step specification	T1
Incorrect relationship	E10: a use case operationalizes a functional feature goal. This is likely incorrect as functional feature goals are normally very abstract and not operationalizable	
Semantic		
Semantic incorrect artifact spec	E11: goal "Users shall be able to write hotels" is incorrect semantically (because *hotels* cannot be *written*)	T2
Incorrect use case specification	E12: a use case has precondition "User has been logged in" and post-condition "Traveller has been signed in." This is invalid given that *User* is equivalent to *Traveller*, "logged in" and "signed in" are synonyms	T3
Inconsistency		
Relationship inconsistency	E13: goal G1 "Users create reviews," G2 "Ensure the reliability of reviews," and G3 "Admins create reviews." G1 requires G2, G2 requires G3, G3 excludes G1. This leads to an inconsistency because it can be inferred that G1 requires G3 while G3 excludes G1	T4
Goal–use case mismatched	E14: goal "users shall be able to create travel articles" is operationalized by a use case with description of "a user edits a review." This is inconsistent because the goal and its associated use case are irrelevant	T3

Table 2 continued

Problem sub-type	Examples	Tnq
Inconsistency involved 2 artifacts	E15: given the goals "Users shall be able to write reviews for tours" and "Travellers shall be able to create reviews for only places." They are inconsistent given *User* and *Traveller* are equivalent while *Tour* and *Place* are disjoint concepts in this domain	
Inconsistency involved more than 2 artifacts	E16: three goals are specified: "if a user account is locked, system sends an email notification to the user" (G1), "if a user account is locked, system sends a SMS notification to the user" (G2) and "if system send a user an email notification, it won't send any SMS notification to that user" (G3). They are inconsistent because it can be deducted from G1 and G2 that both email and SMS notification will be sent in case an account is locked, which is conflicting with G3	T5

Table 3 Problem detection rules used in technique T1

'Invalid Artifact' Rule	**'Invalid Relationship' Rule**
`specifiedBy(a, sfs), elementOf(sf, sfs), not allow(Type(a), sf) => invalid(a)`	`link(a1, a2, rel), not allow(type(a1), type(a2), rel) => invalid(rel)`
`specifiedBy(a, sfs), elementOf(sf1, sfs), elementOf(sf2, sfs), incompatible(sf1, sf2) => invalid(a)`	**'Missing Artifact' Rule** `require(type(a), art_type, rel), not exist(a, art_type, rel) => missing(a, art_type, rel)`

`a, a1, a2` are artifacts. `rel` is a relationship
`specifiedBy(x, y)`: `x` is a list of semantic functions to specify artifact `y`
`allow(x, y)`: semantic function `y` is allowed to specify an artifact of type `x`
`incompatible(x, y)`: semantic functions `x` and `y` are not allowed to be concurrently used to specify an artifact
`link(x, y, z)`: artifacts `x` and `y` are connected by relationship `z`
`allow(x, y, z)`: relationship `z` is allowed to connect an artifact of type `x` and an artifact of type `y` according to meta-model
`exist(x, y, z)`: there exists artifact of type `y` has a `z` relationship with artifact `x`
`missing(x, y, z)`: there is a missing artifact of type `y` to be connected to the artifact `x` by the relationship `z`	

comparison results in the equivalence in *agents* and *activities* and disjoint (only) in *objects*. Table 5 presents the algorithm for detecting inconsistency between two goals.

T4: Rule-based inference The key idea is to use Semantic Web Rule Language (SWRL) (Horrocks et al. 2004) to capture relationships and their rules. Based on that, new relationships can be automatically deduced using Pellet reasoner. In example E13, the relationships are conflicting because: (1) "require" is transitive, and (2) "require" and "exclude" relationships cannot be defined between the same pair of artifacts. The

Table 4 Algorithm to check missing relationships (T2)

```
function checkMissingRelationship(a1, a2)
  a1 and a2 are two goal specifications
  a1_act = getActivity(a1) //extract activity of artifact a1
  a2_act = getActivity(a2)
  a1_mos_act = getMOSDesc(a1_act) //transform to MOS form
  a2_mos_act = getMOSDesc(a2_act)
  rel = findPossibleRel(a1_mos_act, a2_mos_act)
  if(rel != null && !exist(rel))
    report missing relationship
```

Table 5 Algorithm to detect goal pairwise inconsistency (T3)

```
function checkGoalPairwiseInconsistency(a1, a2, expl)
  a1 and a2 are two goal specifications
  expl is inconsistency explanation, is NULL initially
  sfs1 = getSemanticFunctions(a1) //get all semantic functions
of a1. Verb and object are combine into an activity function
  sfs2 = getSemanticFunctions(a2)
  if sfs1 has at least an element which is not in sfs2 AND
vice versa
    return false
  common_sfs = getCommonSfs(sfs1, sfs2) //common set of seman-
tic functions of two a1 and a2
  conflicting_sfs = new empty List of semantic functions
  overlapped_sfs = new empty List to semantic functions
  for each sf in overlapped_sfs
    sf_val1 = getSemanticFunctionValue(a1, sf)
    sf_val2 = getSemanticFunctionValue(a2, sf)
    sf_mos_val1 = getMOSDesc(a1, sf)
    sf_mos_val2 = getMOSDesc(a2, sf)
    if(!relevant(sf_mos_val1, sf_mos_val2))
      return false
    if(isConflicting(sf_mos_val1, sf_mos_val2))
      conflicting_sfs.add(sf, sf_mos_val1, sf_mos_val2)
    else if(isOverlapping(sf_mos_val1, sf_mos_val2))
      overlapped_sfs.add(sf, sf_mos_val1, sf_mos_val2)
  end for
  if(count(conflicting_sfs) == 1)
    expl = generateInconsistencyExpl(conflicting_sfs, over-
lapped_sfs)
    return true
  return false
```

relationships are represented in SWRL as *require*(*G1*, *G2*), *require*(*G2*, *G3*) *and exclude*(*G3*, *G1*). This problem can be detected given the above rules are also captured. Table 6 presents the relationship rules in our framework.

Table 6 Rules of relationships in GUI-F (T4)

require(x, y), require(x, not y) => invalid require(x, y), exclude(x, y) => invalid exclude(x, y) => exclude(y, x) refine(x, y) => require(y, x) require(x, y), require(y, z) => require(x, z) constrain(x, y) => require(y, x)	x, y are goals

Table 7 Algorithm to converting goals with conditions/events (T5)

```
function extractConditionEvent(goals)
  goals is an independent copy of list of goals in the model
  addedArtifacts = new empty map
  for each goal g in goals
    if(hasConditionOrEvent(g))
      split g into g1 and g2, with g1 is the condition/event
and g2 is the rest of g
      setRelationship(g1, g2) //either require or exclude
      addedArtifacts.add(g1, g); addedArtifacts.add(g2, g)
  end for
  for each artifact a in addedArtifacts
    duplicate = false
    for each goal g in goals
      if(isDuplicate(a, g))
        duplicate = true
        mergeRelationships(a, g)
        break
    end for
    if(!duplicate)
      goals.add(a)
  end for
```

T5: Relationship extraction This technique is used to detect inconsistency between more than two artifacts that usually forms an inconsistent cyclic relationship between artifacts specifications. This type of problems normally occurs between goals. In example E16, the inconsistency is detected by separating the main descriptions and conditions or events in a specification into different statements with "require" or "exclude" relationships. For instance, the first specification is converted into "a user account is locked" requires "system sends an email notification to the user." Doing so would transform this problem into the relationship inconsistency problem in which the technique T4 can be applied. Table 7 presents the algorithm for such conversion.

6.2 Measuring goal–use case model's quality

The following formulas are used to determine the completeness, consistency and correctness scores of the model.

$$\text{completeness rate} = \frac{\text{total}_{\text{artefacts and rels}}}{\text{total}_{\text{artefacts and rels}} + \text{total}_{\text{missingartefacts and rels}}}$$

$$\text{consistency rate} = \frac{\text{total}_{\text{artefacts and rels}} - \text{total}_{\text{inconsistentartefactsandrels}}}{\text{total}_{\text{artefacts and rels}}}$$

$$\text{correctness rate} = \frac{\text{total}_{\text{artefacts and rels}} - \text{total}_{\text{incorrect artefacts and rels}}}{\text{total}_{\text{artefacts and rels}}},$$

where $\text{total}_{\text{artefacts and rels}}$ is the total number of artifacts and relationships in the models

- $\text{total}_{\text{missing artefacts and rels}}$ is the total number of artifacts and relationships which are identified as missing by the tool
- $\text{total}_{\text{inconsistent artefacts and rels}}$ is the total number of artifacts and relationships which are involved in the inconsistencies identified by the tool
- $\text{total}_{\text{incorrect artefacts and rels}}$ is the total number of artifacts and relationships that are involved in the incorrectness identified by the tool.

.

6.3 Resolution

6.3.1 Incompleteness resolutions

Incompleteness resolutions come directly from the reasons of the identified problems. For instance, if a specification is missing an object, then "adding an object" is a resolution. If a relationship is missing between two artifacts, then its addition will be suggested.

6.3.2 Incorrectness resolutions

Resolutions for incorrectness are mainly removal or change requests. For instance, if an invalid semantic function is used, the resolution is to remove it. If a use case has its post-condition equivalent to its precondition. Then either the involved precondition or post-condition is requested to be changed.

6.3.3 Inconsistency resolutions

Inconsistency resolutions are based on three strategies: *removal*, *change* and *restriction weakening*. The *Removal* strategy removes one of the conflicting artifacts. The *Change* strategy requires the modifications of the parts of specifications that cause inconsistency. The *Weakening* strategy is used when conflict arises from restrictions in one or more specifications. For instance, *G1*: "Users shall be able to write reviews for tours" and *G2*: "Users shall be able to create reviews for only places." This strategy suggests the removal of the "only" restriction in G2.

7 Usage example

We have developed a tool, GUITAR (Goal and Use case Integration Tool for the Analysis of Requirements) (Nguyen et al. 2014), to support our modeling and analysis process in GUI-F. GUITAR can be downloaded from http://goo.gl/gCUofM. In this section, we use

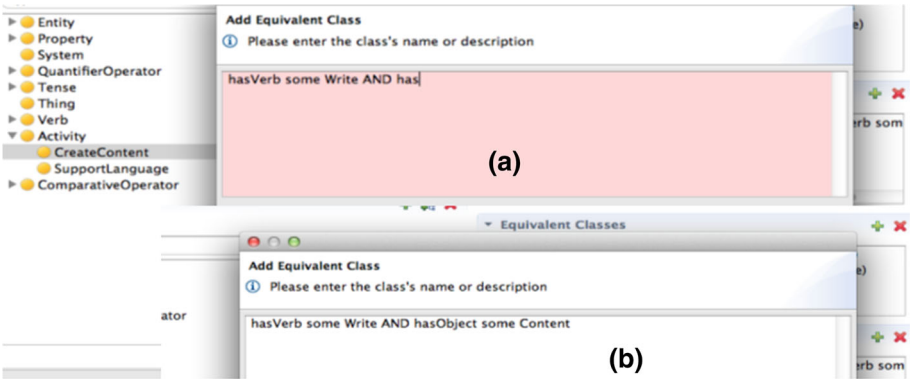


Fig. 8 Updating the domain ontology with ontology editor

the motivating scenario in Sect. 2 to illustrate our approach of modeling goals and use case; and detecting and resolving incorrectness, incompleteness and inconsistency.

7.1 Ontology update

GUITAR incorporates an ontology editor that allows users to quickly maintain and extend ontologies on the fly. Assume that the requirements engineers have obtained ontology for the traveler social network domain. They now want to update it with an *equivalent* relationship between the activities "create content" and "write content." Figure 8a shows that such relationship is being added. GUITAR shows a light red background if the input is invalid or incomplete; and white otherwise (Fig. 8b).

7.2 Adding artifacts

GUITAR enables users to enter artifact specifications in natural language. To facilitate automated analysis, besides textual descriptions, GUITAR requires their structured descriptions that can be transformed into Manchester OWL Syntax for automated reasoning. To do this, GUITAR provides a database of boilerplates used to specify textual descriptions. This boilerplate database was built by us and collected from the work of Hull et al. (Hull et al. 2005) and Cesar project (Rajan and Wahl 2013). In structured specifications, each term is mapped to an ontological concept if it exists in the ontology. If the term does not exist in the ontology, GUITAR suggests an addition of such term. The mapping is done by first using Wordnet (Fellbaum 1998) to get the standard form (called *stem*) of the term used (e.g., *searches*, *places* have their stems as *search* and *place*, respectively), then verify if such stem exists in the ontology.

Assume that the requirements engineers are adding a new functional service goal: "FSG30—users shall be able to create reviews for only places." To do this, they open a "new artifact" dialog, selects to create a functional service goal. GUITAR then generates the goal's ID based on the previously used IDs. Figure 9a shows that the boilerplate "<agent > shall be able to < action > < object>" is being selected for specifying FSG30. Figure 9b depicts the input of the textual description. It is "<Users > shall be able to < create > < only reviews for places>." Figure 9c shows the structured description generated by GUITAR: "Agent(Head(*User*)) + Verb(Create) + Object (Quantifier(QuantifierOperator(Only)) +

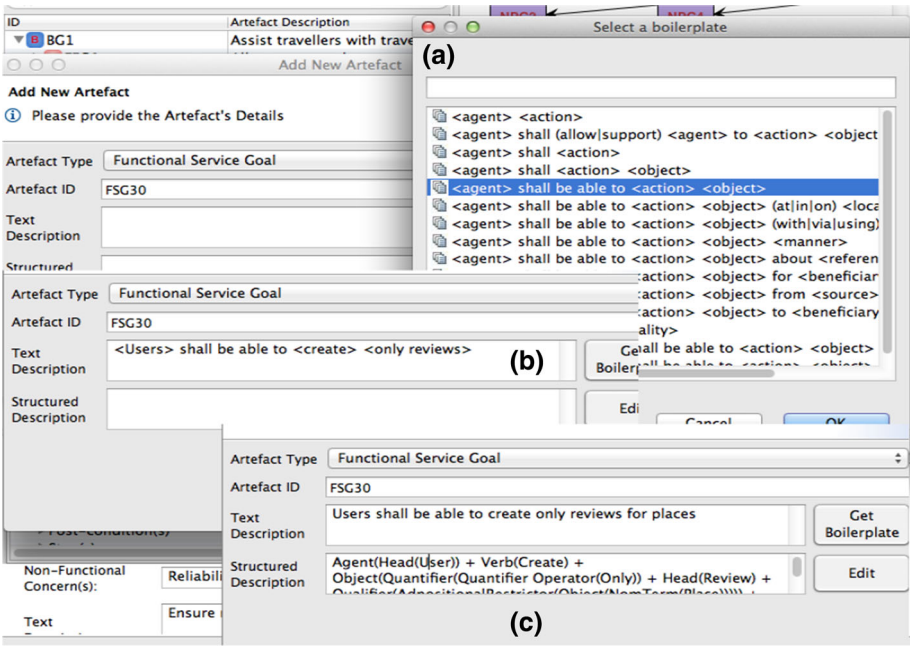


Fig. 9 Adding an artifact

Head(Review) + Qualifier(AdpositionalRestrictor (Object(NomTerm(Place))))) + Negation(NO) + Tense(Present)."

In this specification, all terms within the parenthesis are concepts in the given ontology. This automated transformation is done by mapping the list of parameters in the selected boilerplate to the list of corresponding semantic functions in Functional Grammar (i.e., the *agent* tag < *Users* > is transformed into *Agent*(*Head*(*User*)) which is a simplified form of *Agent*(*NomTerm*(*Head*(*User*))). GUITAR knows that a *nominal term* is used to describe this *agent* because "*User*" is an *entity* in the ontology and that semantic function *agent* can only be described by a *nominal term* according to our meta-model (if the value used is found not an *entity*, GUITAR can raise an error). If the term does not exist in the ontology, GUITAR invokes Wordnet to check the classification of the term. If it is a noun, then it can be considered as an entity.

For complicated descriptions of parameters (i.e., *only reviews for places*), we use regular expressions together with the concepts in domain ontology to produce the internal structure of the corresponding semantic function's specification. For instance, "only" is identified as a *quantitative operator*, "review" and "place" are *entities* according to the domain ontology. Since this description matches the regular expression "quantitative_operator[]*(\d)?[]*entity for entity," it can be transformed into *Quantifier*(*QuantifierOperator*(*Only*)) + *Head*(*Review*) + *Qualifier*(*Adpositional Restrictor* (*Object*(*NomTerm*(*Place*)))).

7.3 Inconsistency detection

The requirements engineers now try to validate the set of artifacts for inconsistencies. They start the inconsistency validator. An inconsistency has been identified between *FSG30*—

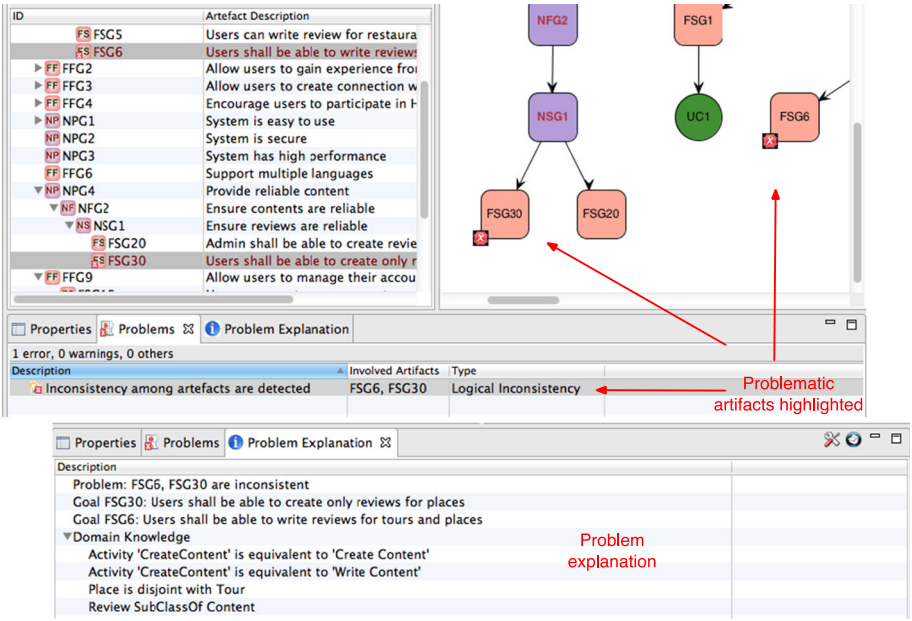


Fig. 10 Inconsistency detected

"*users shall be able to create reviews for only places*" and *FSG6*—"*users shall be able to write reviews for places and tours.*" Figure 10a shows that the problem is described and the involved artifacts are highlighted in both artifact tree view and graphical view. The explanations (Fig. 10b) show that they are conflicting as "create content" is equivalent to "write content," "review" is a subclass of "content," and "place" is disjoint with "tour."

7.4 Inconsistency resolution

The requirements engineers now want to resolve the detected inconsistency. They click on "Quick Fix" icon and GUITAR provides a dialog showing a number of possible resolution options (Fig. 11). They include deleting either FSG30 or FSG6, remove the "only" restriction in FSG30 (weakening strategy), make other modifications to either FSG30 or FSG6, or ignore the problem for now and come back later.

8 Evaluation

In this section, we describe an evaluation using three industrial case studies in the domains of traveler social network (TSN), online publication system (OPS) and split payment system (SPS)[7] and three others from PROMISE (PRedictOr Models in Software Engineering) dataset (Boetticher et al. 2007). The PROMISE dataset consists of 625 requirements collected from 15 software development projects. Among them, 255 items are

[7] Except requirements from TSN are confidential, original requirements for OPS and SPS can be found at http://www.cse.msu.edu/~chengb/RE-491/Papers/SRSExample-webapp.doc and http://www.cise.ufl.edu/class/cen3031sp13/SRS_Example_1_2011.pdf.

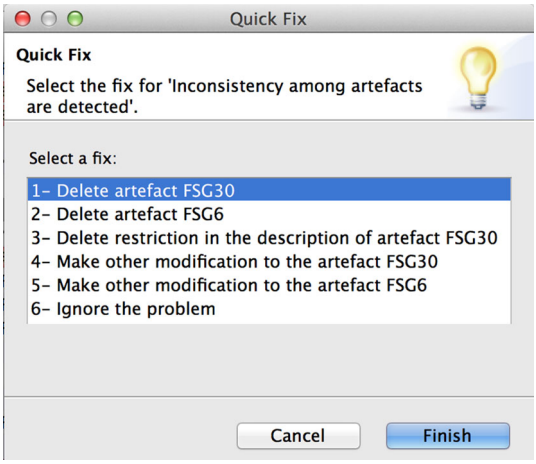


Fig. 11 Inconsistency resolution

marked as functional requirements and the remaining 370 non-functional requirements items are classified into 11 categories, such as Security, Performance and Usability. In this evaluation, we have selected requirements sets from three projects in the domains of Master Scenario Events List Management (MSEL), Real Estate (REs) and Nursing Training Program Administration (NTPA). The numbers of artifacts in these PROMISE case studies are smaller compared with those in the previous three case studies because they do not contain use cases and business goals, due to the focuses mainly on functional and non-functional requirements of this dataset.

The purpose of our evaluation is twofold: (1) to evaluate the suitability of our goal–use case integration meta-model for capturing integrated goal and use cases requirements models and (2) to evaluate the effectiveness of our approach in detecting 3Cs problems. The data used in our evaluation can be found at http://goo.gl/gCUofM.

8.1 Meta-model evaluation

This evaluation was aimed at assessing the suitability of our meta-model for capturing and representing goals and use cases in the mentioned case studies. In this evaluation, we performed two steps: Firstly, for each artifact, we verified if it could be categorized into one of our defined artifact classes in the meta-model (i.e., business goal). Secondly, we verified if that artifact could be represented using our specification rules in the meta-model's specification layer. Table 8 summarizes the results of modeling the case studies. It shows that our meta-model is suitable for capturing goals and use cases since we successfully categorized every artifact into one (and only one) artifact class.

However, we encountered some problems with temporal properties in artifact specifications, i.e., a goal "System logs a user off after 15 min of being inactive" or a constraint "A locked account is locked until an admin unlocks it" are not specifiable in GUI-F. In addition, specifications with uncommon time expressions like "The product shall be available for use 24 h per day, 365 days per year" are also not supported. At this stage, our automated analysis does not deal with time and temporal specifications, and thus, the meta-model was not designed to capture these properties. We leave this to future research.

Table 8 Meta-model evaluation results

	BG	FFG	FSG	NPG	NFG	NSG	UNC	DC	UC	Tot.
TSN										
C	4	11	23	4	7	14	29	9	21	122
NC	0									0
NS	0	0	0	0	1	4	7	0	0	12
OPS										
C	2	14	27	4	6	11	19	5	11	99
NC	0									0
NS	0	0	0	0	2	3	3	0	0	8
SPS										
C	2	16	21	5	11	9	24	11	16	115
NC	0									0
NS	0	0	0	0	2	2	6	0	0	10
MSEL										
C	0	0	15	5	2	5	5	0	0	32
NC	0									0
NS	0	0	0	0	0	2	0	0	0	2
REs										
C	0	2	15	11	6	4	2	0	0	40
NC	0									0
NS	0	0	0	2	0	0	0	0	0	2
NTPA										
C	0	6	37	9	8	3	0	13	0	76
NC	0									0
NS	0	0	0	2	0	0	0	0	0	NS

C classified, *NC* not classified, *NS* not specifiable, *BG* business goal, ***FFG*** functional feature goal, *FSG* functional service goal, *NPG* non-functional product goal, *NFG* non-functional feature goal, *NSG* non-functional service goal, *UNC* use case non-functional constraint, *DC* data constraint, *UC* use case

8.2 Analysis support evaluation

This evaluation was aimed to assess the effectiveness of our artifact analysis approach. We use precision rate and recall rate to measure the soundness and completeness of the analysis results. The precision metric is used to evaluate the soundness. A higher precision means the approach returns more valid results (true-positive—TP) and less invalid results (false-positive—FP). The recall metric is used to assess the completeness of the approach. A higher recall means the approach returns more valid results (true-positive) and has less missed valid results (false-negative—FN). The maximum of both precision and recall are 1 (Eqs. 1, 2).

$$\text{Precision} = \frac{\text{Valid Detected Problems (TP)}}{\text{Total Detected Problems (TP + FP)}} \tag{1}$$

$$\text{Recall} = \frac{\text{Valid Detected Problems (TP)}}{\text{Total Detected Problems (TP + FN)}} \tag{2}$$

Table 9 Effectiveness results of our approach (with PROMISE data)

Problem types	MSEL	REs	MTPA	Precision	Recall
Incorrectness					
D	6	5	20	100 %	100 %
FP	0	0	0		
FN	0	0	0		
Incompleteness					
D	11	7	11	86 %	83 %
FP	1	1	2		
FN	2	1	2		
Inconsistency					
D	0	0	0	–	–
FP	0	0	0		
FN	0	0	0		

D detected problems, *FP* false-positive, *FN* false-negative

8.2.1 Evaluation with PROMISE data

We started with the three PROMISE projects' requirements. For each project, we built an ontology for storing knowledge and semantics in its domain. The ontologies were checked for quality anomalies before the evaluation. In this evaluation, if a requirement could be classified as a GUI-F artifact, it was then entered into GUITAR to build a goal–use case model. Since PROMISE data mainly contain a list of requirements without explicitly specifying their connections, artifact relationship validation was ignored in this experiment.

The evaluation was done by first using GUITAR to identify the problems and the result was then manually verified by us. Table 9 shows the result of this evaluation.

The main problems found in these case studies were incorrect goal specifications. For instance, "The product shall synchronize with the office system every hour" is defined as a non-functional goal instead of a functional goal. Both precision and recall for incorrectness detection in this evaluation are 100 %. We achieved 86 and 83 % for incompleteness evaluation's precision and recall, respectively. That was due to GUITAR's suggestion about a missing non-functional goal of a specific type (i.e., *law compliance*) when it is not needed. We noticed that it is not feasible to thoroughly verify the recall rate of incompleteness detection since it is not possible to find all potential missing requirements for a project. Our verification was done by determining missing requirements based on the domain knowledge that we collected and stored in the relevant ontologies. Although no inconsistency existed in the case studies, there was a positive result that GUITAR did not identify any "problem" which actually was not a problem (false-positive is zero).

8.2.2 Evaluation with other industrial case studies

Due to the limitation of the range of problems existing in the PROMISE data; we carried out another round of evaluation with the other industrial case studies (TSN, OPS and SPS). In this round, we took correct requirements and manually introduced incompleteness, inconsistencies and incorrectness into the set of artifacts. Doing so enabled us to determine if there was a problem that could not be detected by our tool (false-negative), or an identified problem was not actually a problem (false-positive). The seeding process was

Table 10 Approach's effectiveness evaluation results (with other case studies)

Problem types	TSN	OPS	SPS	Precision (%)	Recall (%)
Incorrectness					
D	9	12	7	100	88
FP	0	0	0		
FN	2	1	1		
Incompleteness					
D	21	22	24	85	86
FP	3	2	5		
FN	2	4	3		
Inconsistency					
D	8	8	5	100	88
FP	0	0	0		
FN	1	1	1		

D detected problems, *FP* false-positive, *FN* false-negative

done by first referencing example 3Cs problems from the literature, categorizing them into different sub-types, each sub-type was then classified into different difficulty levels (called *sub-type/difficulty level categories*) by investigating how sophisticated the relevant detection techniques are. For instance, under the "use case specification incorrectness" sub-type, "invalid use case step specification" is categorized as a *simple* problem as most of them can be detected by meta-model matching technique (and some simple ontology inference), while problems of "Different use case exceptions whose conditions are same/subsuming each other" are considered *difficult* since the inference over multiple ontology concepts and relationships may be needed for identifying these problems. The last step of this process was to introduce similar problems into the case studies. For each type of 3Cs problems (i.e., incorrectness), we maintained the balance between the numbers of problems belonging to different sub-type/difficulty level categories. Specifically, 66 incompleteness, 32 incorrectness and 24 inconsistency problems were introduced across three case studies. When generating problems, we considered only those involving artifacts that can be specified in GUI-F (i.e., without temporal properties in their specifications).

Table 10 shows that GUITAR achieved 100 % for precision and 88 % for recall in incorrectness detection. The main kinds of incorrectness detected were mismatches between artifacts descriptions and their categories. GUITAR could not reveal some artifacts that were placed on invalid levels of abstraction. For instance, in the context of a traveler social network system, if a *functional service goal* is defined as "users shall be able to configure their settings," then it has an incorrect specification since "configuring settings" is a very abstract activity and cannot be directly operationalized into a use case. It instead should be defined as a functional feature goal and refined by more specific (functional service) goals, for instance, "Users shall be able to configure their profile privacy settings." GUITAR could not identify this since it was unable to recognize abstract activities from more specific ones. We plan to overcome this issue by extending GUITAR with an automated artifact abstraction classifier.

We achieved 85 and 86 % for precision and recall, respectively, in incompleteness identification. That was due to our use of non-functional constraint categories in identifying missing non-functional artifacts, similar to the problem discussed in the evaluation with PROMISE data. We plan to overcome this issue by developing a category of non-functional constraints that are normally used in a number of common domains.

We have achieved 100 % precision in inconsistency detection. However, the recall rate is 88 %. This was because GUITAR currently does not support the identification of related words used in different forms. For instance, GUITAR was not able to detect an inconsistency between two constraints "uploadable images must be less than 4mb" and "users shall be able to upload images up to 5mb." In this example, it could not find the relationships between the term "uploadable images" and the activity "upload images." We plan to overcome this issue by incorporating a natural language technique to identify the relationships between different forms of a word (i.e., *uploadable* and *upload*) and based on that infer the relationships between different specifications.

8.3 Benchmark validation

In this section, we discuss the benchmark validation that was carried out to evaluate GUITAR's capabilities in analyzing 3Cs problems.

8.3.1 Benchmark applications setup

The selection criteria for benchmark approaches were as follows:

- *Compulsory criteria*
  1. The approach is capable of automatically or semiautomatically analyzing textual requirements for at least one of the 3Cs problems.
  2. The approach's tool can be from either research or industry.
  3. The approach's tool is available to download and installable on common operating systems (Windows, Mac OS and Linux), OR.
     The approach's tool is not available, yet the approach is knowledge based and their demonstrative or validation data are available for reuse with GUITAR.
- *Desirable criteria*
  4. The approach provides support for goal and/or use case modeling and analysis
  5. The approach employs domain ontologies for semantic analysis.

The selection process resulted in seven approaches in which five with tools and two without tools. In the list below, the tool name is used if it is available; otherwise, the name of the first author of the contribution is used.

- *Requirement quality analyzer* (*RQA*) RQA[8] is an industrial tool that uses a wide set of metrics to assess requirements mainly for correctness, consistency and completeness. RQA is based on natural language processing, ontologies and semantic techniques to allow a comparison of the meaning of the requirements.
- *Requirements assistant* (*RA*) Requirements Assistant[9] is an industrial tool that analyzes requirements written in a natural language. It detects incompleteness, inconsistency, vagueness, testability issues and ambiguity in a set of requirements.

[8] http://www.reusecompany.com/requirements-quality-analyzer.

[9] http://www.sirius-requirements.com/product/.

- *Innoslate* Innoslate[10] is an industrial tool that provides an automated support to evaluate the clearness, completeness, design orientation (i.e., whether a requirement describes design option) and verifiability.
- *Requirements-Driven Software Development System (ReDSeeDS)* ReDseeDS (Osis 2010) is a research tool which offers a full Model-Driven Engineering lifecycle which include modeling and analyzing requirements. ReDseeDS is the only tool in this evaluation that supports the modeling of both goals and use cases.
- *Requirements Processing Tool (ReProTool)* ReProTool (Drazan and Mencl 2007) is a research tool that is capable of semiautomatically analyzing use cases written in natural language.
- *Kaiya*: Kaiya and Saeki (2005) verify completeness, correctness and consistency of textual requirements based on ontologies of knowledge and semantics. A tool was reportedly developed for this approach. However, it could not be obtained.
- *Dzung*: Dzung and Ohnishi (2009) proposed a similar technique to Kaika. However, it focuses on providing suggestion on improving the completeness of requirements based on domain ontologies. The tool from this approach was not obtained.

8.3.2 Experimental setup

We divided the benchmark approaches into two groups. The first one contains the 5 approaches with tools and the other two belong to the second group.

For the first group, we used the three above case studies: TSN, OPS and SPS for validation. For each benchmark application, we import requirements from the case studies in the format that the application supports. In case an application did not allow use case specifications, use case steps were extracted and entered as normal requirements. Moreover, in RQA that supported the use of ontologies, we entered our ontologies developed in the previous validation into the tool (in its supported format).

For each approach in the second group, we recreated ontology in our format based on the ontology they provided. For each provided requirement, we determined the artifact type it should belong to (i.e., "Users can retrieve books via the Internet" was classified as a functional service goal) and entered it into GUITAR.

8.4 Experiment results

8.4.1 First group: benchmark approaches with tools

Table 11 depicts the results achieved in this validation. In the following subsections, we discuss the results obtained by each benchmark application in comparison with the results produced by GUITAR. The detailed experimental data and results of all benchmark tools can be found at http://goo.gl/gCUofM.

8.5 RQA

8.5.1 Inconsistency

The majority of inconsistencies detected by RQA were false-positives. There were cases when two unrelated requirements, such as "System shall be secure" and "System shall be

[10] https://www.innoslate.com/.

Table 11 Benchmark validation result for group 1

Case study	Problem type	No. of problems	Detection result by each tool (true-positive/false-positive/false-negative)					
			GUITAR	RQA	RA	Innoslate	ReDSeeDS	ReProTool
TSN	Incorrectness	11	9/0/2	N/A	N/A	0/16/11	0/0/11	0/0/11
	Incompleteness	20	18/3/2	N/A	0/14/20	0/0/20	2/0/18	3/0/20
	Inconsistency	9	8/0/1	2/3267/7	0/2/9	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability problems	Ambiguity problems	N/A	N/A
OPS	Incorrectness	13	12/0/1	N/A	N/A	0/18/13	0/0/13	0/0/13
	Incompleteness	24	20/2/4	N/A	0/12/24	0/0/24	3/0/21	2/0/22
	Inconsistency	9	8/0/1	2/3010/7	0/2/9	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability problems	Ambiguity problems	N/A	N/A
SPS	Incorrectness	8	7/0/1	N/A	N/A	0/11/8	0/0/8	0/0/8
	Incompleteness	22	19/5/3	N/A	0/18/22	0/0/22	2/0/20	2/0/20
	Inconsistency	6	5/0/1	2/3537/4	0/3/6	N/A	N/A	N/A
	Others	N/A	N/A	Other quality problems	Testability and ambiguity problems	Ambiguity problems	N/A	N/A

easy to use," are flagged as inconsistent. Moreover, a number of inconsistencies detectable by GUITAR, however, remained unknown in RQA.

8.5.2 Incorrectness

The requirements marked as having low quality by RQA were those contain the word/phrase "shall" or "shall be able to" and those with abstract concepts such as "easy to use," "quickly." However, since GUITAR is designed to deal with goals on different abstractions level and goals are usually described in "shall be able to" sentences, we accept these vagueness characteristics. Therefore, it was not applicable to compare GUITAR and RQA regarding incorrectness detection.

8.5.3 Incompleteness

RQA evaluates the completeness of requirements by matching a requirement description with its list of boilerplates. Since the boilerplates used in RQA are different from the boilerplates used in GUITAR (which are designed for goal and use case specifications), a comparison between the tools was not suitable.

8.6 Requirements assistant (RA)

All inconsistencies identified in TSN, OPS and SPS case studies were false-positives. For instance, two requirements "system shall store the list of articles" and "system shall store the list of reviewers" were considered inconsistent by the tool while they obviously are not. While having no support for correctness evaluation, RA validated the completeness of requirements based on a set of *absolute* keywords such as *all*, *any*, *anything*. For instance, the requirement "When a new bill is entered into the group, *all* group members will be notified by email" was flagged as incomplete with a suggestion of reviewing for exceptions by the tool. We consider such cases should be better classified as general warnings rather than actual completeness problems.

8.7 Innoslate

Innoslate identified 16, 18 and 11 incorrectness cases in the TSN, OPS and SPS case studies, respectively. However, they were all false-positives according to our judgment. For instance, "Members shall be able to view their debts" was flagged as design-oriented statement, while it is a requirement describing a functionality of the system. Moreover, there was no incompleteness reported in all case studies.

8.8 ReDSeeDS

ReDSeeDS was able to identify use cases with missing preconditions or a use case extension in which no *resume* step or *failure* step defined. However, it was unable to identify a majority of problems in the case studies. For instance, invalid relationships (i.e., both "operationalizing" and "conflict" relationships defined between the same pair of requirements) or a use case with semantically identical pre and post-condition.

Table 12 Benchmark validation result for group 2

Problem type	Tools			
	Kaiya	GUITAR	Dzung	GUITAR
Incorrectness	2	0	0	0
Incompleteness	2	2	11	8
Inconsistency	1	1	0	0

8.9 ReProTool

Due to the lack of semantic support, ReProTool could not identify semantic problems. Moreover, only simple syntactic problems were identified. More complicated problem, i.e., a lack of an extension to handle the case when the article file does not exist in the use case step "The editor attaches the article file to the form" was not detected.

8.9.1 Second group: approaches without tools

Table 12 presents the results obtained for this group. For Kaiya and Dzung, the value in each cell depicts the number of problems of the corresponding types detectable by their approaches (according to their papers). For GUITAR, it shows the number of problems detected by our tool.

GUITAR could identify all 3Cs problems by Kaiya except 2 incorrectness cases. In Kaiya, if there is a requirement containing at least a word which is not mapped to an ontological item, then it is considered incorrect. For example, the requirement "User can play a music in any speed" is incorrect since its words are not fully mapped to ontological items. This requirement, however, was not considered incorrect by our tool since it is not malformed by itself. It was flagged as a warning instead.

In the comparison between Dzung and GUITAR, our tool identified 8 over 11 incompleteness cases raised by Dzung. The other three cases were suggested by Dzung to enhance the requirement completeness because there exist the corresponding functions in the ontology. For example, a requirement about adding a user into the system is suggested since "add user" function exists in the ontology. GUITAR did not raise an attention regarding this function because there were no existing requirements that describe relevant functions to "add user." The rationale is that domain ontology may be very large and reused across projects in the same domain. A single function may be required in a project but may not be needed in another. Therefore, considering a function defined in the ontology that has no corresponding requirement specified as incompleteness could overwhelm users with a large number of false-positives. Thus, from the results of this validation, it was shown that GUITAR was able to identify at least all representative problems detectable by Kaiya and Dzung.

8.10 Threats to validity

8.10.1 Internal threats

Firstly, the types of requirements problems exist (for evaluation on PROMISE data) or being seeded in each case study may vary and affect the precision and recall rates. For instance, if there are more problems of the types the tool cannot detect and less problems of

the types it can identify, then the precision will be reduced, and vice versa. To alleviate this, we tried to generate situations in which the problems are realistic and the balance between different problem types is maintained. In the evaluation with seeded problems, we referenced example 3Cs problems in the literature to ensure the seeded errors were realistic. Moreover, we maintained the balanced numbers of errors across categories and difficulty levels. In future work, we plan to carry out an evaluation with industry partners' real requirements and errors.

Secondly, the quality of domain ontologies might affect the validity of our analysis support evaluation (Sect. 8.2). However, since this validation was designed to validate the soundness and completeness of our technique based on what it "knew" (what was stored in the ontology) and what it "did not know" (what was not stored in the ontology), the quality of ontology is less important. In fact, in this validation, we did not judge the tool as unsatisfied for not detecting a problem whose relevant semantic and knowledge are not known to it. Therefore, the cases that mattered were those in which GUITAR failed to identify a problem while it had sufficient knowledge to detect such problem. Following this principle, we have minimized the effect of ontology quality by using an iterative process in which we ran the validations, then checked the results with the set of the generated problems, then verified if there were any problems that had not been identified because of missing concepts or relationships in the ontologies. If yes, then updated the ontologies with those missing parts and ran the validations again. If no, we recorded that our tool had failed to identify the problems and diagnosed the reasons. The result from the last iteration was the most important one because in that iteration we could eliminate all cases in which a problem was undetected when the tool had no relevant knowledge about it. The documented result in the paper was the one after the last iteration.

8.10.2 External threats

Firstly, in the case study-based evaluations, having good results in these case studies might not imply good results in others. To alleviate this, we selected six case studies in different domains with different types of requirements and most importantly, ensured problems in those case studies covered a wide range of 3Cs problems. Secondly, in our benchmark validation, having good results with the selected approaches may not imply similar results with others. However, the best attempt has been made to get all relevant tools that were available to download and install at the time the validation was carried out. In addition, to alleviate this threat, we have broadened our selection criteria to consider approaches with no available tool but having their demonstrative data available for use to validate GUITAR.

9 Discussion and future work

Our GUI-F framework proposes a set of techniques for modeling and analyzing goal and use case models. Within the framework, the Goal–Use Case Integration meta-model provides a structure and guidelines for modeling goals on different levels of abstraction and their operationalizing use cases. The functional grammar-based structured descriptions of those artifacts enable the artifacts to be parameterized in a consistent way. Such parameterization allows the artifact descriptions to be transformed into formal descriptions in MOS that then promotes the automated analysis of the artifacts. Importantly, GUI-F is

designed to encapsulate "intelligence" so that it can deal with problems that require domain specific knowledge to analyze. Such "intelligence" includes knowledge and semantics in certain domains that are to be captured and represented in the form of ontologies. The principle behind this feature is that the more GUITAR "knows," the higher analysis quality it can perform. This allows GUI-F to be further augmented by enriching the domain ontologies.

Our experiments to date showed some promising results regarding the capability of GUITAR in analyzing 3Cs problems. Our benchmark validation indicated that GUITAR could identify a number of 3Cs problems that were not detectable by other tools. The key reasons are the use of domain ontologies and the technique for parameterizing requirements consistently. This second point explains for the better outcome GUITAR achieved in the validation compared with other ontology-based tools. For instance, in RQA, false-positive values were very high since requirements were compared with each other based on the meanings of the words they contained without considering which roles played by those words (i.e., object, beneficiary). Regarding user experience, while Kaiya and Dzung tools require manual matching of requirements and ontological concepts, it can be done semi-automatically in GUITAR based on the use of our structured descriptions.

In the followings, we discuss the key limitation and challenges of our framework.

9.1 The effort of building domain ontologies

Ontology building is a time and labor-intensive exercise. It is an iterative and ongoing process rather than a one-time task. In order to create a domain ontology, a domain study needs to be performed. In such study, domain concepts and relationships between them are extracted. In addition, knowledge and constraints needs also to be captured. In our view, the effort required for creating ontologies is a challenge rather than a limitation of the framework. It can be considered as a trade-off between the ability of identifying semantic problems, that are not detectable without a collection of domain knowledge and semantic (such as an ontology), and the effort spent to create such ontology. In fact, ontology-based approaches have been used extensively in RE and software engineering. According to a recent systematic literature review on the applications of ontologies in RE (Dermeval et al. 2015), several ontology-based techniques have been proposed and proven the effectiveness of using ontologies in solving different problems in RE. As shown in the benchmark validation, our approach was able to identify problems that are not detectable by other tools, partly because the use of domain ontologies. In addition, according to the same study, 34 % of the reviewed approaches reported their success in reusing ontologies in their contributions. In our view, reusability is one of the key benefits of ontologies. Creating ontologies can be costly; however, it can be reused across projects in the same domain.

We plan to build an "initial ontology" which contains pre-populated concepts that are commonly shared across domains, as an extension of our collection of commonly used verbs. It is intended to be done by consulting lexical resources like Wordnet (Fellbaum 1998), Verbnet and other dictionaries. Such "initial ontology" would potentially help reduce the time and effort since only very domain specific semantics and knowledge would need to be captured in the ontology creation process.

9.2 The lack of support for temporal properties in requirements

Our approach currently does not support the specification of temporal properties such as "until," "unless," "after x seconds" due to two reasons. Firstly, these properties are not

supported by functional grammar. In functional grammar, all these properties are captured as a general "time" semantic function. However, that does not sufficiently indicate the semantic difference between these properties. In addition, temporal properties are not specifiable by MOS, the language used for formal specifications of artifacts in our work. It is planned to consider extending functional grammar with additional semantic functions to accommodate temporal properties. Each property would be associated with a unique semantic function so that its semantic role in a specification can be fully differentiated. In addition, we plan to investigate the possibility of enhancing MOS with a set of inference rules concerning temporal properties in our future work.

9.3 The manual selection of boilerplates for transforming textual artifact specifications into structured specifications

Currently, GUITAR supports a semiautomated way of representing artifacts in structured specifications. In this process, a manual selection of suitable boilerplate for a textual specification is needed. It is our intention to minimize manual effort by automating the transformation of textual artifact specifications into structured ones. It is intended to use the Stanford lexicalized parser (Klein and Manning 2003) to analyze the part-of-speech of words and their dependencies in a specification. This information could provide a starting point to identify the suitable semantic role a word, or a group of words, should belong to. Based on this, the identification of semantic functions in a specification can be supported.

10 Related work

The key-related research approaches could be categorized into two groups: goal and use case coupling approaches and requirements automated analysis techniques.

10.1 Goal–use case coupling

Cockburn (1997) structures scenarios by connecting every action in a scenario to a goal assigned to an actor. He defines three dimensions of goal refinements and relationships between goals and use cases. However, non-functional goals are kept out of scope, and thus, it is not clear how they are verified. Syntactic and semantic analysis of goals and use cases are not supported.

Rolland et al. (Rolland et al. 1998) focus on guiding the elicitation of goals using scenarios based on the concept of *requirement chunks*. Each chunk contains a goal and a scenario in which the scenario operationalizes its associated goal. There are three abstraction types of requirement chunks: contextual, system interaction and system internal level and four abstraction types of goals: business, design, service and internal goals. At each level of requirement chunks, a goal is operationalized into a scenario, and thus, a new chunk is formed. Rolland et al. also provide writing guidelines for goals and scenarios based on Functional Grammar (Dik 1989), though these guidelines are still very high level and do not specifically define how each artifact is specified and verified. Kim et al. (2006) extend (Rolland et al. 1998) to provide guidelines for transferring goals and scenarios into use case models. These approaches provide no analysis support to detect 3Cs problems.

Van Lamsweerde and Willemet (1998) propose a way to use scenarios as typical examples of system usage in KAOS framework. Yet, it lacks support for artifact

specification rules and thus does not validate specifications syntactically. Semantic verification of goals and use cases is also not provided.

In GBRAM (Anton et al. 2001) scenarios are used as a way to uncover hidden goals or obstacles. This framework provides a number of heuristics to help with the elicitation of scenarios and analysis of goals. However, it does not allow both syntactic and semantic verification of artifact specifications.

10.2 Automated requirements analysis

Much work has been done in automated analysis of requirements, ranging from formal methods, natural language techniques to ontology-based techniques. However, few address problems specific to the integration of goals and use cases. Limited work has been done on detecting conflicts between goals and associated use cases or missing relationships between goals and use cases.

10.2.1 Formal methods

While providing comprehensive support for requirements analysis, a major practical challenge of using many formal methods is the use of complicated formal languages and/or logics. KAOS framework (Van Lamsweerde 2001) supports the management of requirements conflicts. In KAOS, requirements are formalized in linear temporal logic (LTL) to enable the conflicts detection using formal reasoning methods. However, the use of LTL requires requirements engineers to be sufficiently familiar with LTL and be able to correctly specify requirements. Thus, its applicability in practice can be fairly limited (Dwyer et al. 1999). In addition, although KAOS supports the creation of domain knowledge through the use of its conceptual meta-model, concepts and relationships in the problem domain and the semantics of requirements cannot be adequately expressed and reasoned in LTL (Breaux et al. 2008).

Tropos (Fuxman et al. 2004) provides model-checking-based analysis of early requirements. Tropos relies on the i* modeling language and the KAOS temporal specification language and thus share the complexity problem of KAOS. Moreover, although supporting inconsistency detection between goals, both frameworks do not provide explicit automated supports to detect inconsistency between goals and use cases and incorrectness and incompleteness among the artifacts.

Lee et al. (1998) proposed a method to formalize use case specifications using Petri nets in order to automatically identify inconsistency and incompleteness of use cases. This work, however, include the complicated formalization process of use case and the lack of semantic capturing which makes semantic problems undetectable.

Sikora et al. (2010) developed a technique to automatically verify the consistency between scenarios across abstraction levels using message sequence charts. However, it does not deal with semantic errors and other problems such as incorrectness and incompleteness among goals and use cases.

10.2.2 Natural language-based techniques

These techniques use natural languages as the principle representation language for requirements. Initially, requirements are input using natural language and then transformed into logics or other forms for automated reasoning. Lami et al. (2004) proposed an

approach with QuARS tool to analyze textual requirements specifications. QuARS is able to identify linguistic defects in requirements. However, the analysis support is limited to syntax-related issues, while morphological and semantic-related problems are not directly addressed.

Gervasi and Zowghi (2005) support automated inconsistency detection in high-level natural language requirements. In this work, natural language requirements are automatically transformed into propositional logic statements to identify inconsistencies. This, however, does not deal with semantic-related issues. 3Cs problem analysis in the context of goals and use cases is also not supported.

10.3 Ontology-based techniques

Ontology-based requirements analysis approaches rely on the use of domain ontologies that capture domain knowledge and semantics together with OWL languages to provide automated reasoning support. In our previous work (Nguyen et al. 2012, 2014), we developed a method to identify inconsistencies, overlaps and redundancies in high-level goals based on the formalization of requirements in Manchester OWL Syntax (MOS). The approach allows for the integration of domain knowledge and semantics into the analysis and therefore makes it possible to identify semantic-related issues. However, this work does not support the detection of incompleteness and incorrectness. In addition, the MOS-based requirements representation language is more suitable for high-level goals and insufficient to specify low-level requirements with more details (i.e., use case interaction steps). Moreover, it does not provide support for analysis goal–use case integration models.

Kaiya and Saeki (2005) proposed an approach of using domain ontologies to detect inconsistency, incompleteness and incorrectness in requirements. Each term in a requirement is mapped to an ontological concept for getting semantic support. The analysis is done based on the relationships between concepts in the ontology. For instance, if a requirement is linked to a concept "create," if "create" has a "require" relationship with "edit" and there is no requirement linked to the concept "edit," then there is an incompleteness. The disadvantage of this approach is that using relationships between atomic concepts may not be sufficient to produce complete and sound analysis results. For instance, if there are requirements about "create an account" and "edit a review." It is not detectable that a requirement about "edit an account" is missing. In addition, the approach does not provide analysis support in the context of goal–use case integration. Cesar (Rajan and Wahl 2013) supports analyzing natural language requirements using domain ontologies. Its tool support allows users to enter natural language requirements, which are then automatically transformed into some representations to allow automated reasoning. This provides similar ontological support to Kaiya and Saeki and thus shares the same disadvantages.

RAT (Verma and Kass 2008) is an ontology-based requirements analysis tool that allows requirements to be input in natural language. Based on a set of standardized requirements specification syntax and user-defined glossaries, it enforces requirements to be written in a systematic way to avoid linguistic defects (i.e., ambiguity, terminology inconsistency). In addition, the tool offers a way to identify potential conflicting requirements by categorizing requirements into different pre-defined ontology classes whose relationships are captured. RAT, however, lacks support for the detection of logical inconsistency, inconsistency involving more than two artifacts and several goal–use case models' specific types of incompleteness and incorrectness.

11 Summary

We have described a new goal and use case integration framework (GUI-F) aimed at complementing existing goal and use case coupling approaches with automated analysis support for incorrectness, incompleteness and inconsistency. At the central of the framework is our developed meta-model for goal–use case integration. The meta-model consists of the artifact layer that provides classification of different artifacts in GUI-F and their various relationships and a specification layer that defines specification rules for all artifacts. In GUI-F, the functional grammar-based parameterization of artifact specifications and domain ontologies facilitate the detection of both syntactic and semantic 3Cs problems. We developed GUITAR, a tool that automates the analysis process. GUITAR allows textual requirements to be entered using boilerplates and automatically transforms them into structured specifications that are then used for automated reasoning. GUITAR also generates comprehensive explanations and resolutions for detected problems. Experiments with six industrial case studies showed that our meta-model is suitable to capture and model goals and use cases. In addition, we achieved high precision (95 % on average) and recall rates (87 % on average) that indicate high effectiveness of our approach. Moreover, the benchmark validation result showed that our tool could detect a wide range of 3Cs problems that were not identifiable in other approaches.

Acknowledgments The authors gratefully acknowledge support from the Victorian Government under the Victorian International Research Scholarships scheme, Swinburne University of Technology, and the Australian Research Council under Linkage Project LP130100201.

References

Anton, A. I. (1996) Goal-based requirements analysis. In *Requirements engineering, 1996, proceedings of the second international conference on* (pp. 136–144): IEEE.

Anton, A. I., Carter, R. A., Dagnino, A., Dempster, J. H., & Siege, D. F. (2001). Deriving goals from a use-case based requirements specification. *Requirements Engineering, 6*(1), 63–73.

Boetticher, G., Menzies, T., & Ostrand, T. (2007) *The PROMISE repository of empirical software engineering data*. West Virginia University, Department of Computer Science.

Breaux, T. D., Antón, A. I., & Doyle, J. (2008). Semantic parameterization: A process for modeling domain descriptions. *ACM Transactions on Software Engineering and Methodology (TOSEM), 18*(2), 5.

Cimiano, P., & Völker, J. (2005). Text2Onto. In A. Montoyo, R. Muńoz & E. Métais (Eds.), *Natural language processing and information systems* (pp. 227–238). Berlin, Heidelberg: Springer.

Cockburn, A. (1997). Structuring use cases with goals. *Journal of Object-Oriented Programming (JOOP/ROAD), 10*(5), 56–62.

Dermeval, D., Vilela, J., Bittencourt, I. I., Castro, J., Isotani, S., Brito, P., et al. (2015). Applications of ontologies in requirements engineering: A systematic review of the literature. *Requirements Engineering*, 1–33. doi:10.1007/s00766-015-0222-6.

Dik, S. C. (1989). *The theory of functional grammar*. Berlin: Walter de Gruyter.

Drazan, J., & Mencl, V. (2007). Improved processing of textual use cases: Deriving behavior specifications. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack & F. Plášil (Eds.), *SOFSEM 2007: Theory and practice of computer science* (pp. 856–868). Berlin, Heidelberg: Springer.

Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering. Proceedings of the 1999 international conference on, 1999* (pp. 411–420): IEEE.

Dzung, D. V., & Ohnishi, A. (2009). Improvement of quality of software requirements with requirements ontology. In *Quality software. QSIC'09. 9th international conference on,* 2009 (pp. 284–289): IEEE.

Fellbaum, C. (1998). *WordNet*. London: Wiley Online Library.

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

Gervasi, V., & Zowghi, D. (2005). Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and Methodology (TOSEM), 14*(3), 277–330.

Glinz, M. (2000). Improving the quality of requirements with scenarios. In *Proceedings of the second world congress on software quality* (pp. 55–60).

Horridge, M., Drummond, N., Goodwin, J., Rector, A. L., Stevens, R., & Wang, H. (2006). The manchester OWL syntax. In: *Proceeding of of the OWL Experiences and Directions Workshop (OWLED'06) at the ISWC'06*.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., & Dean, M. (2004). SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member submission*, *21*, 79.

Hull, E., Jackson, K., & Dick, J. (2005). *Requirements engineering* (Vol. 3). Berlin: Springer.

Kaiya, H., & Saeki, M. (2005). Ontology based requirements analysis: Lightweight semantic processing approach. In *Quality Software*, *2005.(QSIC 2005). Fifth international conference on*, 2005 (pp. 223–230): IEEE.

Kim, J., Park, S., & Sugumaran, V. (2006). Improving use case driven analysis using goal and scenario authoring: A linguistics-based approach. *Data and Knowledge Engineering, 58*(1), 21–46.

Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st annual meeting on association for computational Linguistics-Volume 1*, *2003* (pp. 423–430): Association for Computational Linguistics.

Lami, G., Gnesi, S., Fabbrini, F., Fusani, M., & Trentanni, G. (2004). *An automatic tool for the analysis of natural language requirements*. Pisa: Informe técnico, CNR Information Science and Technology Institute.

Lee, W. J., Cha, S. D., & Kwon, Y. R. (1998). Integration and analysis of use cases using modular Petri nets in requirements engineering. *Software Engineering, IEEE Transactions on, 24*(12), 1115–1130.

Nguyen, T. H. (2014). GUI-F Meta-model descriptions. http://www.it.swin.edu.au/personal/huannguyen/dls/GUIF_metamodel.pdf.

Nguyen, T. H., Grundy, J., & Almorsy, M. (2014). GUITAR: An ontology-based automated requirements analysis tool. In *Requirements engineering conference (RE), IEEE 22nd International, 2014* (pp. 315–316): IEEE

Nguyen, T. H., Vo, B. Q., Lumpe, M., & Grundy, J. (2012). REInDetector: A framework for knowledge-based requirements engineering. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering, 2012* (pp. 386–389): ACM.

Nguyen, T. H., Vo, B. Q., Lumpe, M., & Grundy, J. (2014). KBRE: A framework for knowledge-based requirements engineering. *Software Quality Journal, 22*(1), 87–119.

Noy, N. F., & McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Stanford knowledge systems laboratory technical report KSL-01-05 and Stanford medical informatics technical report SMI-2001-0880.

Noy, N. F., & Musen, M. A. (2003). The PROMPT suite: Interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies, 59*(6), 983–1024.

Osis, J. (2010). *Model-driven domain analysis and software development: Architectures and functions: Architectures and functions*. Hershey: IGI Global.

Poveda-Villalón, M., Suárez-Figueroa, M. C., & Gómez-Pérez, A. (2012). Validating ontologies with oops! In A. ten Teije, J. Völker, S. Handschuh, H. Stuckenschmidt, M. d'Acquin, A. Nikolov, N. Aussenac-Gilles & N. Hernandez (Eds.), *Knowledge engineering and knowledge management* (pp. 267–281). Berlin, Heidelberg: Springer.

Rajan, A., & Wahl, T. (2013). *CESAR: Cost-efficient Methods and Processes for Safety-relevant Embedded Systems* (Vol. 978-3709113868): Springer.

Rolland, C., Souveyet, C., & Achour, C. B. (1998). Guiding goal modeling using scenarios. *Software Engineering, IEEE Transactions on, 24*(12), 1055–1071.

Sikora, E., Daun, M., & Pohl, K. (2010). Supporting the consistent specification of scenarios across multiple abstraction levels. In R. Wieringa & A. Persson (Eds.), *Requirements engineering: Foundation for software quality* (pp. 45–59). Berlin, Heidelberg: Springer.

Sommerville, I. (2011). *Software engineering* (9th ed.). Pearson Education Inc.

Sutcliffe, A. (2003). Scenario-based requirements engineering. In *Requirements engineering conference*, 2003. *Proceedings. 11th IEEE international*, *2003* (pp. 320–329): IEEE.

Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Requirements engineering*, *2001. Proceedings. Fifth IEEE international symposium on, 2001* (pp. 249–262): IEEE.

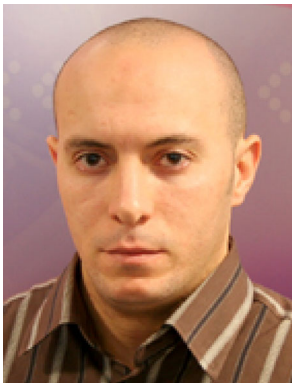
Van Lamsweerde, A., Darimont, R., & Letier, E. (1998). Managing conflicts in goal-driven requirements engineering. *Software Engineering, IEEE Transactions on, 24*(11), 908–926.

Van Lamsweerde, A., & Willemet, L. (1998). Inferring declarative requirements specifications from operational scenarios. *Software Engineering, IEEE Transactions on, 24*(12), 1089–1114.

Verma, K., & Kass, A. (2008). *Requirements analysis tool: A tool for automatically analyzing software requirements Documents*. Berlin: Springer.

Tuong Huan Nguyen is currently a third-year PhD student in the Faculty of Science, Engineering and Technology at Swinburne University of Technology. His doctoral work focuses on automated support for the elaboration and analysis of goal–use case models in Requirements Engineering. He holds a bachelor degree in Business Information System and Honours degree in Computer Science (First Class) both from Swinburne University of Technology. He can be contacted at: huannguyen@swin.edu.au.

John C. Grundy is currently Professor of Software Engineering and Dean of the School of Software and Electrical Engineering at Swinburne University of Technology, Melbourne, Australia. He is Associate Editor in Chief of the IEEE Transactions on Software Engineering, and an Associate Editor for the Automated Software Engineering Journal and IEEE Software. His current interests include domain-specific visual languages, model-driven engineering, large-scale systems engineering, and software engineering education. He is a member of the IEEE and the IEEE Computer Society. More details about his research can be found at: http://www.swinburne.edu.au/science-engineering-technology/staff-profiles/view.php?who=jgrundy.



Mohamed Almorsy is a senior research fellow at Swinburne University of Technology, Melbourne, Australia. Mohamed is current interests include automated software engineering, adaptive security, formal methods, high-performance computing, visualization, and data science. He has a strong track record of research publications in top-ranked international journals and conferences. Mohamed also has more than 10 years of software industry experience. More details about his research can be found at: https://sites.google.com/site/mohamedalmorsy.