From Stakeholder Requirements to Formal Specifications through Refinement

Feng-Lin Li[1], Jennifer Horkoff[2], Alexander Borgida[3], Giancarlo Guizzardi[4], Lin Liu[5], John Mylopoulos[1]

1: Dept. of Information Engineering and Computer Science, University of Trento, Trento, Italy
2: Centre for Human Computer Interaction Design, City University, London, UK
3: Department of Computer Science, Rutgers University, New Brunswick, USA
4: Computer Science Department, Federal University of Espírito Santo, Vitória, Brazil
5: School of Software, Tsinghua University, Beijing, China

Abstract. [Context and motivation] Stakeholder requirements are notoriously informal, vague, ambiguous and often unattainable. The requirements engineering problem is to formalize these requirements and then transform them through a systematic process into a formal specification that can be handed over to designers for downstream development. **[Question/problem]** This paper proposes a framework for transforming informal requirements to formal ones, and then to a specification. **[Principal ideas/results]** The framework consists of an ontology of requirements, a formal requirements modeling language for representing both functional and non-functional requirements, as well as a rich set of refinement operators whereby requirements are incrementally transformed into a formal, practically satisfiable and measurable specification. **[Contributions]** Our proposal includes a systematic, tool-supported methodology for conducting this transformation. For evaluation, we have applied our framework to a public requirements dataset. The results of our evaluation suggest that our ontology and modeling language are adequate for capturing requirements, and our methodology is effective in handling requirements in practice.

Keywords: Requirements Modeling Language, Functional Requirements, Non-functional Requirements, Ontologies

1 Introduction

Stakeholder requirements are notoriously informal, vague, ambiguous, and often unattainable. The requirements engineering problem is to formalize and transform these requirements through a systematic process into a formal, consistent and measurable specification that can be handed over to designers for downstream development. In fact, this is the core problem of Requirements Engineering (RE).

Predictably, there has been much work on transforming informal requirements to a formal specification, going back to the early 90s and before [1][2]. Some of this work exploits AI techniques such as expert systems and natural language processing (NLP) [1]. Other proposals offer a systematic way for formalizing a specification [2]. However, the core problem has not been addressed effectively and has remained open, as attested by current requirements engineering practice, where word processors and spreadsheets continue to constitute the main tools for engineering requirements. For

example, according to a webcast audience poll conducted by Blueprint Software System in 2014, more than 50% of the participants said that they are using documents and spreadsheets for conducting requirements engineering[1]. To address the poor support for collaboration, traceability, and management offered by such vanilla tools, there have been proposals for requirements-specific tools (e.g., Rational DOORS [3]) that support RE-specific activities, such as elicitation, modeling, specification and traceability management. However, these tools pay little attention to the derivation of requirements; instead, they focus more on the management of derived requirements.

Our work attacks the problem afresh, making the following contributions:

- Offers a comprehensive *ontology of requirements*, which consists of various kinds of goals: functional, quality and content goals (descriptions of world sates, i.e., properties of entities in the real world). In addition, our *specifications* include functions (aka tasks), functional constraints, quality constraints, state constraints (machine states that reflect world states) and domain assumptions.
- Proposes a requirements modeling language that can capture the kinds of requirements identified in our requirements ontology, as well as interrelations between them. We also provide a methodology for refining informal stakeholders requirements into formal specifications.
- Presents a three-pronged evaluation of our proposal using a prototype supporting tool and the PROMISE requirements set [4]. First, we classify the whole set of requirements according to our ontology in order to evaluate its coverage; second, we encode all the requirements in the set using our language to assess its adequacy; third, we apply our methodology to two case studies from the dataset, where formal specifications were derived from informal requirements for a *meeting scheduler* and a *nursing scheduler* exemplar[2].

The rest of the paper is structured as follows: Section 2 reviews related work, Section 3 outlines our research baseline, Section 4 presents our requirements ontology, Section 5 sketches the language for capturing requirements, Section 6 presents a methodology (including refinement operators) for deriving formal specification from informal stakeholder requirements, Section 7 presents the three-pronged evaluation, Section 8 summarizes contributions and offers a glimpse of forthcoming work.

2 Related Work

The transformation from informal requirements to formal specifications has been the subject of research for more than 25 years. Early work by Fraser et al. [5] proposed guidelines for developing VDM specifications from Structural Analysis (mostly Data Flow Diagrams). Giese at al. [6] tried to relate informal requirements (UML use case) to formal specifications written in Object Constraint Language (OCL). Seater et al. [7] have discussed how to derive system specifications from Problem Frame descriptions through a series of incremental steps (problem reduction). These approaches focus on functional requirements (FRs), and pay little attention to non-functional requirements (NFRs).

[1] http://www.blueprintsys.com/lp/the-business-impact-of-poor-requirements/

[2] Due to space limitations, only the meeting scheduler case study is presented in this paper. The two complete case studies are available at http://goo.gl/GGceBe.

KAOS [2] constitutes a landmark goal-oriented methodology for deriving formal operational specifications from informal stakeholder requirements. In KAOS, goals elicited from stakeholders are formalized using Linear Temporal Logic (LTL), refined to sub-goals through a set of refinement patterns, and operationalized as specifications of system operations (pre-, post- and trigger conditions) by following a set of formal derivation rules [8]. This transformation process has been extended by many other researchers for deriving formal system specifications from KAOS goal models, as in [9]. The KAOS methodology does facilitate the derivation of functional system specification from stakeholder goals; however, it does not offer support for specifying and refining NFRs, and does not address ontological considerations for requirements.

The NFR Framework (NFR-F) [10] was the first proposal to treat NFRs in depth. NFR-F used softgoals (goals with no clear-cut criteria for success) to capture NFRs. Softgoals have the syntactic form "*type* [*topic*]" (e.g., "accuracy [account]", where "accuracy" is a type and "account" is a topic). The framework offers contribution links for linking software design elements to softgoals, and several operators for decomposing softgoals. Our work builds on these ideas, but aims to offer a comprehensive set of concepts for modeling and analyzing all requirements, not just NFRs.

Quality quantification has been used repeatedly to make NFRs measurable. In this regard, ISO 9126-2 [11] proposed a rich set of metrics for quantifying various quality attributes, while the P-language [12] suggested use of "scale and meters" to specify NFRs. However, these proposals do not offer guidelines or methodologies for deriving formal NFR specifications from informal ones. Techne [13] has proposed operationalizing softgoals into quality constraints which do come with clear-cut criteria for success. Techne facilitates the quantification of softgoals; however, like its NFR-F ancestor, it does not treat well existential dependencies between qualities and functional goals, a distinguishing feature of our proposal.

Ontologies, typically ontologies of specific domains for which requirements are desired, have been employed in RE mainly for activities or processes [14]. These efforts, however, are not proposals for an ontological analysis of requirements notions. In fact, few researchers have attempted to ontologically analyze requirements. Our goal here is in the ontological classification and conceptual clarification of different requirement kinds. In this spirit, the work that is strongly related to ours and receives the most attention in the literature is the Core Ontology for RE (aka CORE) [15]. Our work proposed in [16] and continued here is in line with CORE in several aspects. For instance, both proposals are founded on the premise that requirements are stakeholder goals and that NFRs should be interpreted as requirements that refer to qualities. However, there are also important differences between the two proposals. Firstly, CORE is based on the DOLCE foundational ontology, and ours is built on UFO [17]. As discussed in [16], UFO offers a richer set of categories to cover some important aspects of the RE domain, especially regarding the analysis of functional and quality requirements (as shown in Section 4). Secondly, CORE contains a number of deficiencies in handling NFRs [16]. For instance, it is unable to capture a class of requirements that refer to both function and quality, or neither qualities nor processes/events (in ontological term, perdurants), but entities (endurants), and it does not favor the expression of requirements that are vague but do not refer to qualities.

3 Research Baseline

This work builds on our recent work on quality requirements (QRs) [16][18], where we proposed an ontology for classifying, a formal language for modeling, and some refinement operators for refining QRs. Our existing requirements ontology (as shown in the unshaded part of Fig. 1) is based on a goal-oriented perspective where all requirements are goals of one sort or another. That ontology, however, focuses on quality goals (QGs) and quality constraints (QCs) that are used to capture quality requirements (QRs). These constitute the most important class of what has been traditionally called non-functional requirements (NFRs). The difference between a QG and a QC is that the former is vague while the latter comes with a clear-cut criterion for success.

According to [18], we treat a quality as a mapping from its subject to a quality region, and define a QR as a QG that requires a quality to map into values within a region QRG. Therefore, we write a QG as *Q* (*SubjT*): *QRG*, a syntactic abbreviation for ∀*x. instanceOf* (*x*, *SubjT*) → *subregionOf* (*Q*(*x*), *QRG*), meaning that for each individual subject *x* of type *SubjT*, the value of *Q*(*x*) should be a sub-region of (including a point in) *QRG*. Note that the subject of a quality is not limited to an entity, function/task or process, but can also be a goal, as well as a collective of entities or processes (e.g., as in *"90% of all executions shall be within 5 sec."*).

Using this syntax, the requirement "*the product shall return* (*file*) *search results in an acceptable time*" can be captured as in Eq. 1.2. *Quality constraints* (QCs) that operationalize QGs use the same syntax, but must have a measurable region (see Eq. 1.3). For more interesting examples please refer to [16][18].

search' := search <actor:{the product}><object: file> (1.1)

QG1-1 := processing time (*search'*) *: acceptable* (1.2)

QC1-2 := processing time (*search'*) *: ≤ 8 sec.* (1.3)

In addition to the syntax, we provide operators for refining QGs/QCs, including *relax* and *focus*. *Relax* is used to make a requirement practically satisfiable or alleviate inconsistency between requirements. Specifically, we use ***U*** (universality), ***G*** (gradability), and ***A*** (agreement) operators to relax practically unsatisfiable requirements. For example, we weaken "*all the runs of file search*" to "*x*% *of the runs*" by using U, relax "*within 8 sec*." to "*nearly within 8 sec*." by using G, or relax "(*all the*) *web users shall report the UI is simple*" to "*y*% *of the web users*" by using A. *Focus* offers two ways to refine a QG: via the quality *Q* based on reference quality hierarchies (e.g., ISO/IEC 25010 [19]) or via the subject type *SubjT* according to the parts of an entity or the functional goal hierarchy. Take a "*security*" QG in a software development process for example, in the former case, stakeholders may lay particular emphasis on one of its sub-qualities, say "*integrity*"; in the latter case, we may not need to secure the entire system (e.g., the interface) but some important parts (e.g., the data transfer module).

4 An Ontology for Requirements

In this section, we extend the ontology of NFRs in our previous work [18] to a full-fledged ontology for requirements, with a focus on functional and content requirements. Our classification criteria is based on fundamental concepts such as *function*, *quality and subject* (the bearer of a quality or function), along with the ontological

semantics of the Unified Foundational Ontology (UFO) [17]. In general, both *functions* and *qualities* are existentially dependent characteristics that can only exist by inhering in their *subjects* (bearers). For example, the product search function or the reliability of an e-commerce website would depend on that specific system. Roughly, a *quality* is always manifested as long as it exists. In contrast, a *function* (capability, capacity) is ontologically a *disposition*, and is only manifested when certain *situations* hold. Function manifestations amount to happenings of *events* that bring about *effects* in the world.

In UFO, most perceived events are *polygenic*, i.e., when an event is occurring, there are a number of dispositions of different participants being manifested at the same time. For example, a manifestation event (i.e., a run) of the product search function will involve the capacities of both the system and a user. In software development, we can design the capacities of the system (a search function), but often make assumptions about the capacities of the user (e.g., the user is not visually impaired, the user masters a certain language). These kinds of requirements will be captured as functional goals and domain assumptions in our requirements ontology, respectively.

An overview of our extended requirements ontology is shown in Fig. 1, with new concepts shaded. A goal can be specialized into a functional goal (FG), quality goal (QG) or content goal (CTG), to be discussed in detail later. Note that a goal may belong to more than one category, such as FG and QG (e.g., *"the system shall collect real-time information"*), or FG and CTG (e.g., *"... display students records, which include ID, name, GPA, etc."*). When this is the case, a goal is refined into FG and QG sub-goals, or FG and CTG ones. As in [18], a goal can also be operationalized by domain assumptions (DAs), which are assumptions about the operational environment of the system-to-be. E.g., "*The system will have a functioning power supply*".

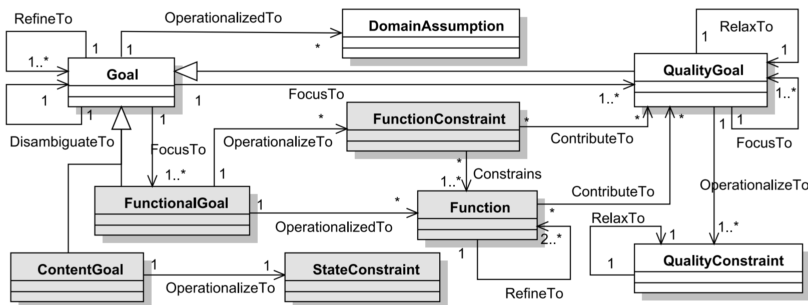

Fig. 1 The extended requirements ontology (based on [18])

Functional Goals, Functions and Functional Constraints. A *functional goal* (FG) represents a requirement that is fulfilled through one or more functions. Following the ontological underpinnings of our approach, an FG would come with the following associated information: (1) *function* — the nature of the required capability; (2) *situation* — the conditions under which the function can be activated; often this includes pre-conditions (characterizations of the situation), triggers (the event that brings about that situation), but also actors (agents), objects, targets, etc.; (3) *event* — the manifestations or occurrences of the function; (4) *effect* (post-conditions) — situations that are brought about after the execution of the function; and (5) *subject* — the individual(s) that the function inheres in. For example, in the requirement "*the system shall*

notify the realtor in a timely fashion when a seller or buyer responds to an appointment request", the "*notify*" function, which inheres in "*the system*", will be activated by the situation "*when a seller or buyer responds to an appointment request*". Moreover, its manifestation, a notification event, is required to occur *in a timely fashion* (note this is not an effect, but a quality of the notification).

A *functional constraint* (FC) constrains the situation in which a function can be manifested. That is, an FC is usually stated as a restriction on the situation of a function. For example, in the FG "*users shall be able to update the time schedule*", one may impose a constraint "*only managers are allowed to perform the update*".

As we can see in these examples, FGs and FCs cannot be simply taken as propositions, as some goal modeling techniques have it. Rather, they are descriptions. Inspired by this observation, we use an "*attribute*: *descriptor*" language to capture them. E.g., the "*notify realtors*" and the "*update time scheduler*" examples can be captured as in Eq. 2 and Eq. 3, respectively. Note that the curly brackets indicate a singleton, and ':<' denotes description subsumption (e.g., Eq. 3 says that the update function is subsumed by things that only have managers as their actors).

FG2 := *Notify <actor:{the system}><object: realtor><trigger: responds>*
<actor: seller ∨ buyer><target: appointment request>> (2)

FC3:= Update <object: time schedule> **:<** *<actor:* ONLY *manager>>* (3)

Content Goals and State Constraints. Content goals (CTGs) describe desired properties of world states, i.e., properties of entities in the real world. For example, a student in the real world has Id, name and GPA. To satisfy a CTG, the system-to-be must consist of states that reflect such world states. For example, to satisfy the aforementioned CTG, the student record database table of the system must include three columns: Id, name and GPA. Such desired machine states are termed *state constraints*.

Typically, CTGs are needed when defining: (1) data dictionaries, which describe required entities with associated attributes (e.g., the student example above); and (2) multiple objects of a function, e.g., in the requirement "*the system shall display movie title, director, actor, etc.*", there is an implicit concept "*movie detail information*". We show these two examples as in Eq. 4.1 and Eq. 4.2 below.

CTG4-1 := *Student record* :< *<ID: String> <name: String> <GPA: Float>* (4.1)

CTG4-2 := *Movie detail information* :< *<title: String> <director: String>*
<actor: String> (4.2)

Note that although CTG4-1 and CTG4-2 describe a set of attributes (and associated value regions) that should be manipulated by the system, they are not QGs. The key point is that they are not descriptions of *qualities required to be present in the system-to-be*, but rather requirements on *desired properties of entities in the world,* to be fulfilled by the system-to-be.

5 A Requirements Modeling Language

We give the syntax of our language in Fig. 2 using Extended-BNF. Nonterminal are in italics, and terminals are quoted or derived from "…*Name*" nonterminal.

We start with the definition of *Attr*, an "*attribute*: *descriptor*" pair as shown in line 1. An attribute can relate an individual to more than one instance of the description, in which case we can use cardinality constraints "≥ *n*", "≤ *n*", "= *n*", "*n*" or "SOME" (*n*

is a nonnegative integer, "SOME" means "≥ 1"). For example, "<*registerFor*: ≥ 3 *course*>" is a description of individuals who register for at least 3 courses. If the cardinality part is omitted, it is by default "= 1". The keyword "ONLY" implies that the attribute can only have individuals of type described by "*Descriptor*" as fillers. We currently do not provide a built-in set of attributes, which requires an ontology of software systems and of the application domain. That is, we allow engineers to invent new attributes when needed.

```
(01) Attr := '<' AttrName ':' [ [ '≥' | '≤' | '=' ] n | 'SOME' | 'ONLY' ] Descriptor '>'
(02) Descriptor := atomicValue | automicDataType | SubjT
(03) SubjT := Entity | Function | Function'.'AttrName
              | 'NOT' '(' SubjT ')' | SubjT '∧' SubjT | SubjT 'V' SubjT
(04) Entity := EntityName Attr* | Attr+ | '{' IndividualName+ '}'
(05) Function := FuncName Attr*
(06) Goal := GoalName
(07) FG := Function
(08) QG(QC) := QualityName '(' SubjT ')' ':' RegionExpr
(09) RegionExpr := region | QualityName '(' SubjT ')'
(10) CTG(SC) := Entity ':<' Attr+
(11) FC := SubjT ':<' Attr+
(12) DA := SubjT (':<' | '≡') SubjT
```

Fig. 2 The Extended-BNF syntax for our language

The descriptor of an attribute can be an atomic value (e.g., "*Trento*" as the value of address, "5€" as the value of price), atomic data type (e.g., *String*, *Double*, and *Text*), or a subject type ***SubjT*** (line 2). A *SubjT* can be an entity, a function, a filler of an attribute in a function, the negation of a *SubjT*, the conjunction or union of *SubjTs* (line 3). Note that a *SubjT* is a type (or a class in object-oriented terms), and not an individual (instance). If the *SubjT* is a singleton, we wrap it with a curly bracket, e.g., {*the product*}. The constructors '*NOT*, '∧', and '∨' applied to *SubjT* are standard set operations. An example can be "(*active* ∨ *outdated*) ∧ *record*". Note that a function is treated as a type, having its runs as associated set.

An *entity* is composed of an optional entity name and a list of "*attribute*: *descriptor*" pairs, or a set of specific individuals (line 4). An anonymous entity is an entity with omitted name. For example, "<*accessedBy*: *manager*>" represents a type of entity that is accessed by managers. A *function* is represented in a similar way, but must have a function name (line 5). Since a *SubjT* (e.g., an entity or function) itself can be further qualified by attributes, resulting in nested descriptions (a trademark feature of Description Logics). For example, in "*the product shall record all the equipment that has been reserved*", "*equipment*" is the *object* of the function "*record*", and also has an attribute "*status*" (see FG6 in Eq. 6).

*FG*5 := *Protect <actor: {the system}><object: user info ∧ private >* (5)

*FG*6 := *Record <actor: {the system}>*
<object: equipment <status: reserved>> (6)

QG7 := *understandability* ({*the interface*}): *intuitive* (7)

CTG8 := *Non-clinical class :< <course name: String>*
<lecture room requirements: Text> < instructor needs: Text> (8)

In general, when a *goal* has not yet been specialized into sub-kinds like FG, QG or CTG, it can be simply written as a natural language string (Fig. 2, line 6). A *FG* is described as a required function (line 7). E.g., "*user private information shall be protected*" is captured as in Eq. 5. In line 8, a *QG*/*QC* is denoted in the form of "*Q* (*SubjT*): *QRG*" (adopted from our previous work [18]), where *QRG* can be either a *region* (e.g., low, fast, [80%, 95%]) in a value space or an expression that takes value/region from a value space (line 9). For instance, "*the product shall have an intuitive user interface*" is captured as in Eq. 7, in which "*the interface*" is a singleton.

Note that the syntactic form of QGs/QCs enables us to capture the important *inherence* relation between qualities and subjects: by "*Q* (*SubjT*)", we mean the quality *Q* inheres in an individual that is of type *SubjT* and, *SubjT* can be a function, an entity, or a goal. That is, a QG/QC in our framework is able to take a function or an entity involved in a function as its inhering subject. Capturing this inherence relation enables us to better manage QRs and FRs, as shown in our case study in Section 7.

A *CTG* (resp. SC) specifies the world (resp. machine) state of an entity through "*attribute*: *descriptor*" pairs. For example, the CTG "*a non-clinical class shall specify the course name, lecture room requirements, and instructor needs*" is captured as in Eq. 8. We use the subsumption relation (':<') instead of definition ('≡') because the specific entity could also have other properties not characterized at the moment. E.g., a non-clinical class may include extra attributes, such as "*course introduction*". An *FC* is defined in a similar way, but can be imposed on either a function or an entity (type). E.g., "*only managers are allowed to access data tables*" can be captured as "*data table* :< <*accessedBy*: ONLY *manager*>". Finally, a DA assumes a *SubjT* to be subsumed by or equal to another *SubjT*. For instance, the DA "*the system will run on Windows*" can be encoded as "{*the system*} :< <*operation system*: *Windows*>". The definition relation ('≡') in DAs can be used to connect semantically equivalent concepts, e.g., "*list of class* ≡ *sequence of class*".

The semantics of our language can be formalized by translation to a logic that has its own formal semantics already. As an example, interested readers can refer to our technical report available at http://goo.gl/GGceBe for a translation of our language to a Description Logic (DL) language, OWL [20] in this case.

6 A Methodology for Transforming Informal Requirements into a Formal Specification

In this section we first introduce two refinement operators that will be used for refining requirements, and then present a three-staged methodology for transforming informal requirements to a formal specification.

Refinement Operators. In our previous work [18], we have proposed *Relax* and *Focus* to refine QGs. Here we extend the set of refinement operators with *Operationalization* and *Contribution*, to facilitate the transformation process.

Operationalization. In our framework, *operationalization* transforms requirements to specifications. In general, a FG can be operationalized as function(s) and/or FC(s), a QG is operationalized by QC(s), and a CTG is operationalized by SC(s). FGs can be treated as the effect (post-condition) of functions: once the operationalizing functions are performed, the corresponding FG will be satisfied. To operationalize QGs, they must be made measurable — clear quality metrics and value regions must be defined.

E.g., "*good security*" can be operationalized as "*monthly unauthorized access shall be less than* 3". For this purpose, standards like ISO/IEC 9126-2 [11] that have proposed a rich set of metrics for quantifying qualities are helpful. We make use of such standards in our methodology. To operationalize CTGs that describe world states, we need the system-to-be be in certain machine states (e.g., having a certain data base schema). Note that a goal (FG, QG or CTG) can be operationalized by domain assumptions (DAs). That is, it can be assumed true as long as the DAs hold.

Contribution. When QGs are operationalized as QCs, we only have the evaluation or success criteria for corresponding QGs. To meet these criteria, the system-to-be often needs to perform some functions, adopt certain designs, or impose suitable functional constraints. We use the classic contribution links "*help*", "*hurt*", "*make*" and "*break*" from NFR-F, to capture the relations between such functional elements and QGs. E.g., to achieve the "*good security*" QG above, we may need to include in our design functions such as "*authenticate users*" and "*authorize users*" to prevent unauthorized access. Note that contribution links are used to capture relations between functional elements and QGs at design-time, and QCs that measure QGs are evaluated at run-time. E.g., in the above example, at design time we may take that "*good security*" will be satisfied if the two contributing functions are present; at run-time, we need to monitor and check if "*monthly unauthorized access*" is less than 3 times (a QC).

A Three-Staged Methodology. Our methodology consists of three phases: (1) an informal phase, where informal requirements are disambiguated and broken down into goals representing single requirements; (2) a formalization phase, where each informal goal is formalized, along with its relationships to other goals; (3) a smithing phase, where refinement operators are iteratively applied on formally specified goals to derive unambiguous, satisfiable, mutually consistent and measurable specifications.

Informal Phase. Each requirement is treated here as a proposition and can be modeled and refined using existing goal modeling techniques (e.g., Techne [13]). The main tasks of requirement engineers in this phase are to: (1) identify key stakeholder concerns and classify them according to the requirements ontology of Fig. 1; (2) decouple composite concerns to make them atomic, and (3) refine high-level requirements to low-level ones and link functional elements to QGs in the spirit of goal-oriented refinement techniques.

Step 1*: Identify key concerns and classify requirements.* We ask the question "*what does a requirement r concern*?" to determine its classification, and provide some operational guidelines as follows:

- If *r* refers to both function and quality, then it is a composite goal. E.g., "*the system shall be able to interface with most DBMSs*" is composite since it refers to a function "*interface*" and a universality quality "*most*" over the set of DBMSs.
- If *r* refers to only function(s), then it is a FG.
- If *r* refers to only quality(-ies) and is vague (clear) for success, it is a QG (QC) If *r* constrains the situation of a function (e.g., actor, object, pre-condition, etc.), then it is a FC. E.g., "*the students added to a course shall be registered*".
- If *r* makes an assumption about the environment of a system, then it is a DA. For example, "*the product will be used in an office environment*".
- If *r* describes the parts, components or attributes an entity shall possess, then it is a CTG. For example, "*the user interface shall have a navigation button*".

Step 2: Separate Concerns. In case a requirement *r* is a combination of concerns, they need to be separated:

- If *r* is a combination of function and quality, it can be focused into a FG and a QG. E.g., the DBMS example shall be decomposed into a FG "*the system shall be able to interface with DBMSs*" and a QG "*most of the DBMSs*".
- If *r* refers to sibling functions/qualities, it shall be separated such that each resulting requirement concerns one function/quality. E.g., "*the system shall allow entering, storing and modifying product formulas*" shall be decomposed into "*the system shall allow entering …*", "*… storing …*", and "*... modifying …*".
- If *r* refers to nested qualities, we decouple them starting from the innermost layer. E.g., the QG "*at least 90% of the tasks shall be completed within 5 sec.*" can be decoupled into two QGs: QG1 "*processing time within 5 sec.*", and QG2 "*QG1 shall be fulfilled for more than 90% of tasks*" (a universality QG).
- If *r* is a mix of function and content, it is suggested to define a CTG and a FG, respectively. E.g., for "*display date and time*", we will have a CTG that defines an entity "*calendar*" with attributes date and time, and a FG "*display calendar*".
- If *r* includes purposes or means, it shall be decomposed to different goals, which will be connected using refinements or contribution links. E.g., for "*the product shall create an exception log of product problems for analysis*", we will have a FG "*analyze product problems*" being refined to "*create an exception log*".

Step 3: Refine Requirements. In this step, high-level goals are refined to low-level ones by utilizing AND-*refine* or *refine*, and link functional elements to QGs through *contribution* links based on domain knowledge.

Formalization Phase. Here we formalize each goal in accordance with its classification. In the discussion below we focus on FGs, QGs and CTGs. As for other elements such as FCs and DAs, readers can refer to our syntax introduction in Section 5.

- *Functional Goals*. For FGs, we often need to find out its actor, object, and sometimes its target, pre-, post- and trigger conditions. For example, for "*when a conference room is reserved, the scheduler shall be updated*", we can write "*update* <*object*: *scheduler*> :< <*trigger*: *reserve* <*object*: *room*>>".
- *Quality goals*. The three key elements of a QG include quality, subject and desired quality region. Note that the subject can be either a bare function/entity or a complex description. For example, for "*90% of the maintainers shall be able to integrate new functionality into the product in 2 work days*", there are two qualities: "*operating time*" for an integration process and "*universality*" for the set of maintainers. We thus define two QGs: "QG1 := *operating time* (*integrate* <*actor*: *maintainer*> <*object*: *new* ∧ *functionality*> <*target*: {*the product*}>): 2 *work days*", and "QG2 := *U* (*QG*1.*actor*): *90%*".
- *Content Goals*. CTGs require the system-to-be to represent certain properties of entities in the real world. For example, "*the system shall display date and time*" will be captured as a CTG "*Calendar* :< <*hasDate*: *date*> <*hasTime*: *time*>" and a FG "*display* <*actor*: {*the system*}><*object*: *calendar*>".

The encoding process facilitates the detection and resolution of ambiguity: if there is more than one way to encode a requirement, then there is ambiguity. E.g., "*notify users with email*" is ambiguous since it can be mapped into "*notify* <*object*: *user*> <*means*: *email*>" or "*notify* <*object*: *user* <*hasEmail*: *email*>". In such situation, stakeholders have to identify the intended meaning(s).

Smithing Phase. Once goals have been formalized, we iteratively apply refinement operators relax, focus, operationalization, and contribution to derive satisfiable and measurable requirements specifications:

Step 1: Relax. In this step, we analyze whether a requirement is practically satisfiable or not, and use the three operators U, G, A, or a composition thereof to relax a requirement to an acceptable degree. For instance, the requirement "*all the tasks shall be finished within 5 sec.*", captured as "QG1:= *processing time* (*tasks*): *within 5 sec.*", can be relaxed by using G: "QG2 := G (QG1): *nearly*" (all the task shall be *nearly* within 5 sec.), or U: "QG3 := U (QG1): *90%*" (*90% of the tasks* shall be within 5 sec.), or even both "QG4 := U (QG2): *90%*" (90% *of the tasks shall be nearly within 5 sec.*). The A operator is mainly applied to subjective QGs, e.g., "*the interface shall be simple*", captured as "QG5 := *appearance* ({*the interface*}): *simple*", can be relaxed using A as "QC6 := A (QG5): *the majority of surveyed users*".

Step 2: *Focus*. We focus QGs in two ways: via the quality Q or via the subject type *SubjT*. For example, the QG "*usability* ({*the product*}): *good*" can be focused into "*learnability* ({*the product*}): *good*" and "*operability* ({*the product*}): *good*" by following the quality hierarchy in ISO/IEC 25010. These quality goals can be further refined along subject hierarchy, e.g., a meeting scheduler often has functions like set up meeting and book conference room, so the quality "*learnability*" can be further applied to these functions, obtaining QGs "*learnability* (*set up meeting*): *easy*" and "*learnability* (*book conference room*): *easy*".

Step 3: *Operationalization*. In this step, we operationalize FGs as functions, FCs, DAs or their combinations thereof, QGs as QCs, and CTGs as SCs. Our understanding of manifesting events of functions as polygenic enables us to systematically operationalize FGs. Take the example of "*the system shall notify realtors in a timely manner*". What kind of effect is required to satisfy the FG? Is it the case that the goal is satisfied by merely a message being sent by the system? Or, alternatively, does the FG also require the message to be properly received by realtors? In the former case, we only need to design a "*send notification*" function and simply assume certain capacities on receiving (adding one or more DAs). However, in the latter case, we should design both the sending and receiving functions such that the joint manifestations of these functions have the desired quality (i.e., "*timely*"). When operationalizing QGs, vague by nature, to measurable QCs, we suggest using "*prototype values*" [16] to help define quality regions. For example, to operationalize the QG "*the learning time of meeting scheduler shall be short*", we first ask stakeholders "*how long is short*?" Their answers provide prototype values. We can then employ mathematical techniques such as probability distribution or Collated Voronoi diagram, using the obtained prototype values to derive corresponding regions [16]. When operationalizing CTGs, properties of real-world entities being characterized will be mapped to corresponding machine states, often data base schemas. For example, the CTG "*Student* :< <*hasId*: *String*> <*hasName*: *String*><*hasGPA*: *Float*>" will be operationalized as a SC "*Student Record*: <*Id*: *varchar*> <*name*: *varchar*> <GPA: float>".

Step 4: *Contribution*. As discussed, we use contribution links to capture the relations between functional elements and QGs. Note that a functional element may help (make) some QGs but can also hurt (break) others, capturing trade-offs between requirements. For example, "*encrypt data*" can help a security QG while hurting a performance QG. In this case, we can further prioritize QGs through eliciting priorities

from stakeholders [13]. Sometimes stakeholder requirements contain low-level concerns such as "*the system shall be developed using the J2EE runtime library*". In this case, it is necessary to consider refinements from a bottom-up perspective: often we ask "*why*" to elicit the implicit higher-level requirement, e.g., good interoperability with respect to different kinds of operation systems in this example.

7 Evaluation

We present results of our evaluation using the PROMISE (PRedictOr Models in Software Engineering) requirements set, which includes 625 requirements collected from 15 software projects [4]. This dataset comes with an original classification of requirements kinds: 255 items are marked as functional requirements (FRs) and the remaining 370 non-functional requirements items are classified into 11 categories such as security, usability and availability. The counts of original classifications are shown in the second column of Table 1.

The aims of our evaluation are: (1) evaluate the coverage of our requirements ontology by classifying the whole set of requirements; (2) evaluate the expressiveness of our language by formalizing all the 625 requirements using our syntax; (3) illustrate the effectiveness of our methodology by applying it to two case studies from the dataset: *meeting scheduler* and *nursing scheduler*. Due to space limitations, we present only statistics of our evaluation and the meeting scheduler case study here. The complete information of our classification and formal descriptions of the 625 requirements, the two case studies and our technical report can be found at http://goo.gl/GGceBe.

Evaluating our Ontology. We went over the full dataset, identified the key concern of each requirement, and classified them by following the classification guidelines proposed in step 1 of the informal phase. We show our classification counts in Table 1, where we use '+' to indicate a combination of concerns within a requirement (e.g., FG+QG means a mix of FG and QG). These classification results extend the results over the 370 NFRs presented in our previous work [18].

Table 1 The ontological classification of the 625 PROMISE Requirements

Requirements Category	Original	FG	QG	FC	CTG	FG/FC + QG	FG+FC	FG+ CTG	DA
Functional	255	183	6	9	21	1+0	6	29	0
Usability	67	7	46	2	0	11+1	0	0	0
Security	66	11	2	39	0	9+2	3	0	0
Operational	62	14	10	13	0	10+2	6	0	7
Performance	54	3	43	1	0	4+1	1	0	0
Look and Feel	38	9	20	0	1	6+2	0	0	0
Availability	21	0	20	1	0	0	0	0	0
Scalability	21	1	19	0	0	0	1	0	0
Maintainability	17	1	10	3	0	2+1	0	0	0
Legal	13	1	11	1	0	0	0	0	0
Fault tolerance	10	4	4	0	0	2	0	0	0
Portability	1	0	0	0	0	0	0	0	1
Total	625	234	191	69	22	46+9	17	29	8

From each row of table 1, we can see how the original categorization of requirements is distributed across our ontological classification. For example, from the original 255 FRs, we identified 183 FGs, 6 QGs, 9 FCs, 21 CTGs, 1 FG/FC+QG, 6 FG+FC, and 29 FG+CTG. Here 51 out of the 255 (20.0%) of the FRs concern con-

tent. We found that most of the security related NFRs are often FG/FC related (the third row): 97% of them are identified as FGs, FCs, or combination with other concerns (11 FGs, 39 FCs, 11 FG/FC+QG, and 3 FG+FC). For example, for “*only managers are able to deactivate user accounts*” (originally classified as a security NFR, but in fact is an FC), the system needs to check whether the actor is a manager or not when the deactivation function is accessed. One can also see that many requirements (101/625, 16.2%) are a mix of concerns (with ‘+’ in their labels).

Our evaluation shows that FCs, CTGs, and the mix of concerns such as FG+FC, FG +QG, and FG+CTG are not trivial and need more attention in practice. The results also provide evidence that our requirements ontology is adequate for covering requirements in practice.

Evaluating our Language. After classification, we rewrote the set of all 625 requirements using our language to evaluate its expressiveness. In this step, we separated the concerns of a requirement if it was composite, and encoded it by following the guidelines presented in the formalization phase. Our syntax was able to capture all 625 requirements, resulting in 1276 statements (nearly double the amount of original requirements), including 419 FGs/Fs, 313 FCs, 375 QGs, 90 CTGs and 79 DAs. Note that there are 7 instance-level constraints (7/625, 1.12%) identified in our evaluation. We are able to express these constraints by using the “*same_as*” DL constructor [21]; however, the use of “*same_as*” imposes severe limitations on reasoning.

The count of each type of statement in our language does not strictly correspond to the classification counts in Table 1. For example, we have 22 CTG and 29 FG+CTG in Table 1, but ultimately 90 rather than 51 CTGs. This is because the original dataset includes many composite and nested requirements, e.g., sibling functions, nested qualities and content, and we broke these up into separate requirements when encoding them. In addition, we treat domain knowledge as domain assumption(s). For instance, “*Open source examples include Apache web server Tomcat*” was captured as “DA := *Tomcat* :< *web server* ∧ *open source*”.

Our language and guidelines facilitate the identification of ambiguity. During the formalization process, we identified 24 ambiguous requirements (3.84%), and eliminated the ambiguity by choosing the most likely interpretation. For example, “*notify users with email*” will be encoded as “*notify* <*object*: *user*> <*means*: *email*>”. Note that although we could have found some ambiguities by reading natural language requirements text, using a more ad-hoc, less systematic approach, such an approach would likely cause us to miss many ambiguities; as such naïve approaches do not force the user to carefully analyze and classify the text. Furthermore, once ambiguities are found, an ad-hoc approach would not tell us what to do when an ambiguous requirement is found. Our approach provides a systematic way for not only identifying but also dealing with ambiguities in requirements.

Our guidelines also contribute to making requirements accurate and concise. E.g., for a rather informal statement “*the product shall make the users want to use it*”, we can identify its focus by asking the question “*what does it concern?*”, and restate it as a QG “*attractiveness* ({*the product*}): *good*”, which can be further refined, .e.g., “*number of users* ({*the product*} <*period*: *one week after its launch*>) : >= 1000”.

Evaluating our Methodology. We performed two case studies on the meeting scheduler (MS) and nursing scheduler (NS) project, adopted from the PROMISE data set. Here we present the MS case study.

The Meeting Scheduler (MS) project has 74 requirements, including 27 FRs and 47 NFRs. A meeting scheduler is required to create meetings, send invitations, book conference rooms, book room equipment, etc. We classified the 74 requirements according to our ontology, separated the concerns of requirements when needed, encoded them by using our syntax. Next, we refined quality goals using the set of provided operators, including relax, focus, and operationalization, to make them practically satisfiable and measurable. Finally, we obtained a specification, which consists of 67 functions, 67 QCs, 8 FCs, 3SCs, and 10 DAs (155 in total).

We kept the requirements (goals, FGs, QGs and CTGs), specifications (functions, QCs, FCs, SCs and DAs), and the derivation process (refinement, operationalization, contribution, etc.) in a textual goal model, and then translated the whole model to OWL. To support this process, we developed a translation tool based on the OWL API, and used it to systematically and automatically translate the resulting requirements specification into an OWL-ontology.

The major benefit of translating a requirements specification to an OWL-ontology is the convenience of obtaining an overview of concerns, such as quality, function, and entity: we are able to ask a list of questions as shown in table 2 (technically, these questions will be translated into DL queries) [3]. For instance, we can ask "<*inheresIn*: {*the product*}>" (an instantiation of Q2) to retrieve the set of qualities that inhere in "*the product*". Note that these questions are not exhaustive. If desired, we can ask more complex questions like "*what functions are required to finish within 5 sec.*?" in the form of "<*hasQuality*: *ProcessingTime* <*hasValueIn: within 5 sec.*>>".

Table 2 Example useful queries over the requirements specification

ID	Concerned Questions	Syntax
Q1	What kinds of subjects does a quality refer to?	<*hasQuality*: *QualityName*>
Q2	What qualities are of concern for a subject?	<*inheresIn*: *SubjT*>
Q3	Who performs the function?	<*isActorOf*: *FG*>
Q4	What is the function operating on?	<*isObjectOf*: *FG*>
Q5	What functions do a subject is involved in?	<*object*: *SubjT*>

Threats to validity. In our evaluation, the ontological classification of requirements and the encoding of natural language requirements as formal descriptions are performed by experienced modelers (the authors). In the future work, we intend to have others use our requirements ontology and modeling language to confirm their adequacy in capturing requirements. Also, although we have evaluated our requirements ontology and language on only one requirements dataset, the threat to our evaluation is low: (1) the size of the dataset we used is large (including 625 items); (2) the data set is collected by a third-party for software engineering research, hence not biased by ourselves. As for the case study, the meeting scheduler example we used (i) is one of the requirements exemplars for evaluating different kinds of research approaches [22] (ii) is able to demonstrate the different kinds of concepts and operators proposed in our approach (e.g., many of its NFRs need the relaxation and focus refinement; it does include ambiguous requirements that need to be disambiguated). We are also planning to evaluate our framework on industrial examples.

[3] Note that we are not using the full expressive and reasoning power of OWL. We are currently investigating translation to other logics and extending our language to allow more interesting forms of reasoning.

8 Discussion and Conclusions

We propose a framework for transforming informal requirements to formal requirements specifications. Our proposal includes three key contributions to the state-of-the-art: (i) a novel requirements ontology, (ii) a description-based requirements modeling language, and (iii) a methodology (including a set of refinement operators) for transformation purposes.

Our proposal also addresses several important challenges associated with NFRs [23]: (1) NFRs are often vaguely stated and hard to measure; (2) it is hard to specify crosscutting concerns for NFRs; (3) it is difficult to get an overview of NFRs that are associated with a FR; (4) it is not obvious where to document NFRs, etc. Our methodology addresses the first issue. Our treatment of NFRs captures the inherence (existential dependency) between NFRs and FRs, together with our language and tool support, we can easily know what subjects does an NFR refer to and what qualities are of concern with regarding to a subject, thus addressing the second and third issue. In fact, capturing the inherence relation also contributes to resolving the fourth issue: we can define a FR as a subject and relate concerned NFRs with it through the inherence link, turning the whole requirements to a structurally connected graph. In this way, NFRs and FRs are not separated anymore, as they are in the IEEE 830-1998 standard.

Note that our language is designed for requirements engineers rather than stakeholders. Users of our language need to have necessary knowledge and/or need to be trained. Moreover, our approach has limitations on handling temporal constraints. We currently represent temporal constraints with attributes such as "*before*", "*after*", and "*concurrent*". However, the reasoning part of such representations is severely limited. Finally, our language is unable to capture algebraic constraints such as "given an initial balance a, after a withdrawal of b, the balance shall be $a - b = c$".

Several issues remain open, notably inconsistency handling. The resolution of inconsistency may require one to prioritize, relax (e.g., relax the quality region, adding pre-condition) or even drop requirements. This interesting point will certainly be further explored within our framework. Another important issue is how to effectively manage requirements evolution. Currently, we are capturing interrelations between FRs and QRs. It will be very interesting to see how a requirements knowledge base evolves with changing requirements, a major topic in Software Engineering for the next decade.

Acknowledgment. This research has been partially funded by ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution", unfolding during the period of April 2011 - March 2016. It also supported by the Key Project of National Natural Science Foundation of China (no. 61432020), and the Key Project in the National Science & Technology Pillar Program during the Twelfth Five-year Plan Period (No. 2015BAH14F02).

Reference

1. C. Rolland and C. Proix, "A natural language approach for requirements engineering," CAiSE, 1992, pp. 257–277.
2. A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," Sci. Comput. Program., vol. 20, no. 1–2, pp. 3–50, 1993.

3. "IBM - Rational DOORS," http://www-03.ibm.com/software/products/en/ratidoor.
4. T. Menzies, B. Caglayan, Z. He, E. Kocaguneli, J. Krall, F. Peters, and B. Turhan, "The PROMISE Repository of empirical software engineering data," http://promisedata.google code.com.
5. M. D. Fraser, K. Kumar, and V. K. Vaishnavi, "Informal and formal requirements specification languages: bridging the gap," Softw. Eng. IEEE Trans. On, vol. 17, no. 5, pp. 454–466, 1991.
6. M. Giese and R. Heldal, "From informal to formal specifications in UML," UML 2004, pp. 197–211.
7. R. Seater, D. Jackson, and R. Gheyi, "Requirement progression in problem frames: deriving specifications from requirements," Requir. Eng., vol. 12, no. 2, pp. 77–102, 2007.
8. E. Letier and A. Van Lamsweerde, "Deriving operational software specifications from system goals," FSE, ACM SIGSOFT symposium, 2002, pp. 119–128.
9. B. Aziz, A. Arenas, J. Bicarregui, C. Ponsard, and P. Massonet, "From goal-oriented requirements to Event-B specifications," NFM, 2009.
10. L. Chung, B. A. Nixon, and E. Yu, Non-Functional Requirements in Software Engineering, vol. 5. Kluwer Academic Pub, 2000.
11. ISO/IEC, "ISO/IEC TR 9126-2 Software engineering - Product quality - Part 2: External metrics," ISO/IEC, 2003.
12. T. Gilb, Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage. Butterworth-Heinemann, 2005.
13. I. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos, "Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling.," RE, 2010, pp. 115–124.
14. H. Kaiya and M. Saeki, "Using domain ontology as domain knowledge for requirements elicitation," RE, 2006, pp. 189–198.
15. I. J. Jureta, J. Mylopoulos, and S. Faulkner, "A core ontology for requirements," Appl. Ontol., vol. 4, no. 3, pp. 169–244, 2009.
16. R. Guizzardi, F.-L. Li, A. Borgida, G. Guizzardi, J. Horkoff, and J. Mylopoulos, "An Ontological Interpretation of Non-Functional Requirements," FOIS, 2014.
17. G. Guizzardi, Ontological foundations for structural conceptual models. CTIT, Centre for Telematics and Information Technology, 2005.
18. F.-L. Li, J. Horkoff, J. Mylopoulos, R. S. Guizzardi, G. Guizzardi, A. Borgida, and L. Liu, "Non-functional Requirements as Qualities, with a Spice of Ontology," RE, 2014.
19. ISO/IEC 25010:2011, "Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models," 2011.
20. D. L. McGuinness, F. Van Harmelen, and others, "OWL web ontology language overview," W3C Recomm., vol. 10, no. 10, p. 2004, 2004.
21. W. W. Cohen and H. Hirsh, "Learning the Classic Description Logic: Theoretical and Experimental Results.," KR, vol. 94, pp. 121–133, 1994.
22. M. S. Feather, S. Fickas, A. Finkelstein, and A. Van Lamsweerde, "Requirements and specification exemplars," Autom. Softw. Eng., vol. 4, no. 4, pp. 419–438, 1997.
23. R. Berntsson Svensson, T. Olsson, and B. Regnell, "An investigation of how quality requirements are specified in industrial practice," Inf. Softw. Technol., vol. 55, no. 7, pp. 1224–1236, 2013.