Dependency Graph for Requirements Structuring based on Guidance Ontology

N. Darragi and S. Collart-Dutilleul and E.M. El-Koursi and P. Bon

Abstract

The modeling process aims to define and analyse requirements for testing physical scale systems such as robotic surgery machines, railway signalling and control systems, nuclear reactor control systems, etc... These safety-critical systems whose failure may result in severe human or physical damage, are designed to be testable and verifiable before their implementation. To understand the system specifications and its functionalities, different types of models are used and each one reflects a viewpoint of a particular scope. Furthermore, its aim is to risk analysis and performance testing.

Our approach is based on goal-oriented requirements elicitation, structuring and analysis. To automate the latter requirements engineering processes it is crucial that we involve expert knowledge. This process needs to be split into several steps. This work is limited to present techniques to obtain specifications dependency graphs from a set of requirements to determine the global goal model.

This paper presents a domain independent framework for operational requirements modeling beside of specifications structuring technique based on *Guidance* Ontology. The present work focuses on improving the performance of structuring techniques through a pattern recognition based on a guidance ontology. This approach is shown to enable the structuring process automation by making use of domain ontologies as an expert knowledge base and capitalization.

N. Darragi and S. Collart-Dutilleul and E.M. El-Koursi and P. Bon
IFSTTAR-ESTAS, 20 RUE ELISEE RECLUS, BP 70317, F-59666 VILLENEUVE D'ASQ, FRANCE, e-mail: (nesrine.darragi,simon.collart-dutilleul,el-miloudi.el-koursi, philippe.bon)@ifsttar.fr

1 Introduction

The requirements are a non-formal description of user needs or system requirement specifications noted SRS. "It is a statement that identifies a product or process operational, functional, or design characteristics or constraints, which are unambiguous, testable or measurable, and necessary for product or process acceptability (by consumers or internal quality guidelines)." (see [IEEE (1998)]). The interpretation of the requirement specifications may lead to a misunderstanding between development engineers, system analysts and domain experts. In fact, between the real needs of the user, and what the expert expressed and all transformations by modeling techniques and what these techniques offer as an abstraction in a specific viewpoint, there is a risk of having different understandings of the requirements. The ambiguity and the incompleteness of requirements, which lead to the inaccuracy or incorrectness of the system, are mainly due to the difference between the viewpoints. Therefore it is important to share domain knowledge between different stakeholders during the development process. In requirement engineering, the use of ontology's aim is the knowledges standardization [Kaiya et al.(2005)].

Ontologies are used primarily in the context of the Semantic Web [Dieng et al.(2001)] and [Sowa(1999)]. Recently, they have been used in the field of software engineering [Happel et al.(2006)] thanks to their adaptability to semantics and reasoning. Furthermore, they have been used in software maintenance [Dameron(2005)]and requirement engineering.

In contrast to existing works, which use ontologies to generate system specific models, this paper introduces a methodology where, based on specifications in text and diagram forms, where an assisted generator produces a system specific model.

The paper is organized into five sections. After a short introduction, the section 2 describes a domain independent framework architecture for requirements structuring and we briefly introduce our guidance ontology regarding requirements engineering.. The section 3 and the section 4 are devoted to present requirements structuring approach to produce template specifications and control dependency graphs. In the section 5, we discuss an algorithm of dependency graph construction from a flowchart.

2 Domain Independent Framework

There is a need of system specific glossary suited to stakeholders, subsystems and functions collected from the system requirement specifications. We also require domain ontology specific to the system domain in order to instantiate glossary entries over a range of high level concepts and relationships. The figure 1 shows the relationship between system specific knowledge (SRS and glossary), the system domain knowledge (Domain Ontology) and the systems of specification elicitation and structuring.

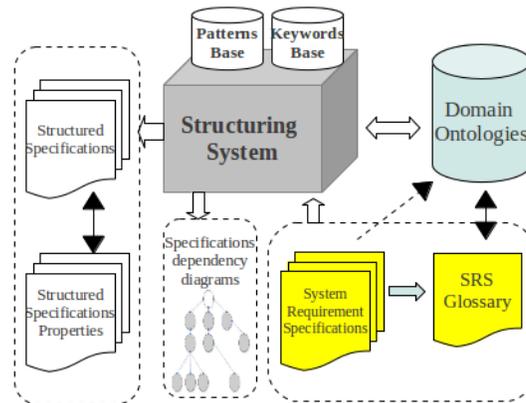Dependency Graph for Requirements Structuring based on Guidance Ontology

Fig. 1 Domain Independant Framework Architecture

As the input of our framework, the customer provides a system requirement specifications and an SRS glossary which is a type of book glossary. This helps the system to capture customer's keywords in specifications. Another important and necessary input is the domain ontologies. Since the framework is an independent domain engine, the system needs specific domain knowledge for each input specification if it is in new domain which the system did not handle before. All input knowledge such as a domain ontology. For example, if we are handling specifications of a railway control system, the framework needs a railway domain ontology. In our case, we propose a guidance ontology of the railway domain to structure railway control system specifications.

2.1 Keywords Base

Our Keywords Base (`KeywordsBase`),is a table that contains words which determine different parts of the sentence such as {shall, while, every, if,...}. These indicators are identified by hand analysing and determining frequently used words and structures. We may identify many groups for these indicators based on its semantics. We propose to extend the classification provided by [**?**] to cover the maximum of requirement structures. The following list indicates possible categories of keywords.

- Imperatives: shall, must, must not, is required to, are applicable, is responsible for, will, should,...
- Continuances: below, as follows, following, listed, in particular, support,...
- Directives: figure, table, for example, note, reference, see section, refer to, following {ref}, e.g,...

- Options: can, may, optionally,...
- Transfer: go to, leads to,...
- Conditionals: if, then, otherwise, once, in preparation, when, with, ...
- Timed: immediately, later, at least, every, after, between, globally, not less than, exactly...
- Weak Phrases: adequate, as a minimum, if practical, as applicable, easy, as appropriate, be able to, be capable, but not limited to, capability of, effective, if practical, normal, provide for, timely, obvisouly, clearly, certainly, some, several, many, etc., and so on, such as, tubed,...
- Other Keywords: although, how, via, but, since, composed of not less than, from, of,...

2.2 The guidance Ontology

To automatically handle the semantics of our system, we need a description of the ontology presented by UML class diagrams. UML Profile is used to extend and customize UML models for a particular purpose of a particular domain.

According to *Ontology Definition Metamodel (ODM)* [OMG(2009)], a Profile UML is proposed in this part to describe our ontology which is an *Ontology UML Profile (OUP)*. Many works use UML and other software engineering techniques not only to develop ontologies in order to use existing advanced tools and standards but also to make ontologies understanding easier.
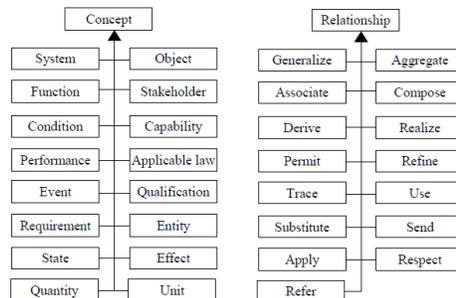

Fig. 2 Concepts and properties of the domain ontology

In figure 2 we present a part of domain ontology. The first diagram defines the concepts and the second shows the relationships between different concepts called properties. We use UML class diagrams for the sake of simplicity, but also because we believe that it is sufficient to present our ontology,i.e. the concepts and relationships between them. We will include specific relations in UML as our list of properties (Generalize, Aggregate, Associate, Compose). This list is needed to show the

dependencies between the different concepts represented as classes. In this work we will not show the specifications hierarchy but this relationships list will be useful in the next step of our methodology. This ontology is specific to a domain so it must be generic so as to enable the hierarchical analysis. The list of properties is enhanced by a standard predefined UML dependency (Derive, Realize, Permit, Refine, Trace, Use, Substitute, Send).

We introduce a new property, called Respect. This is a property necessary in this context to show strong dependencies between requirements. It is used in cases where a requirement must follow a concept, definition, rule, standard or other requirement. The ontology concepts in the field of requirements engineering, including safety requirements are defined in a general context. This allows us to represent the requirement specifications . Properties will be used not only to define the relationships between different concepts in the same requirement, but also to define the links between the requirements and traceability. The meta-model of (Fig. 3)
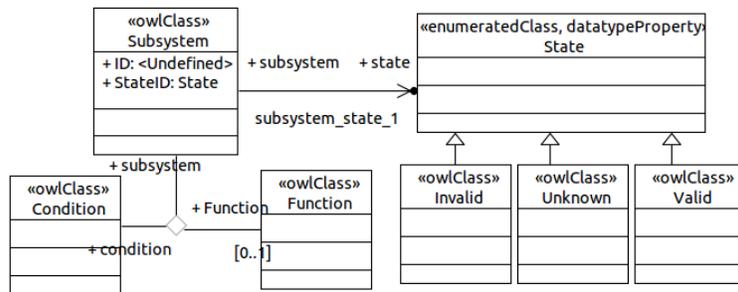

Fig. 3 Part of Guidance ontology Meta-model

3 From requirements to templated specifications

In the first step of algorithm 1, we extract sentences from *SubRS*. Each sentence is split into different parts. If a list of sentence elements are not empty and the sentence parts are unitary, we may resume processing. This is what we referred to in the algorithm as *"understand"*. If it is not possible to *"understand"* the sentence, we ask the assistance of a *SubRS* expert. The `detectKeywords` function uses a table of "keywords" `KeywordsBase`. When keywords are detected, we try to determine the appropriate pattern by `detectPattern`. We use patterns to identify the structure of requirements. We proposed various patterns to handle SubRS by detecting keywords.

We analyse in the next phase of the same step the structure of sentence based on keywords and we classify it as a statement, a conditional sentence (in case of

if-then-else, *when-then*,...) or a transfer sentence (*go to*, *leads to*). Finally, we apply a pattern from `PatternBase` or we create it if we are dealing with a new sentence structure. Example 1 shows a possible structure of a statement. Example 2 shows the structure of a transfer sentence. A conditional sentence is handled by pattern like the one shown in example 3.

This sentence categorization is not the unique or the best classification but we need to know information about dependencies between specifications and about the conditions of transitions between subsystem states.

The detection pattern callable unit shown by algorithm 3, returns a pattern of each statement after detecting instances of every statement element. `detectInstance` uses *SubRS* glossary and places them on `InstStack`. All detected patterns are also placed in the `StructSpec` table.

Example 1. Subsystem **shall** *function object* **every** *performance units*

Example 2. Some Transfer patterns:
Subsystem **leads to** [Requirement Ref]
Subsystem **go to** [Requirement Ref]

Example 3. **If** *subsystem property* **is** *state* **, / then** [`Declarative pattern` Or `transfer pattern`]

If no pattern is detected, we create a new one based on its meta-model. The function is shown by algorithm 2.
In this step, we are unable to identify real attributes such as the real *subsystem* or in the instance of *state* related to the property of the subsystem in example 3. We need a type of knowledge base to help us to determine possible attributes of existing concepts. Patterns contain high level concepts that have to be instantiated for every *SubRS* in a lower level of abstraction. Structuring specifications leads to the separation of domain knowledge from operational knowledge. What we need is a finite list of terms such as a glossary. This will be a *SubRS* related glossary that heritates from a high level light ontology shown in the next section.

4 From Structured Specification to Control Dependence Graph

The second part concerns the construction of a dependency graph from structured segments. This procedure translates the transfer statement (contains **go to** or **leads to** ..) into two nodes connected by directed edge from the *source reference* to *destination reference*. The translation of a structured specification which does not contain a transfer statement is more complicated. When specification is a conditional or declarative statement, a semantic analysis is necessary.

Dependency Graph for Requirements Structuring based on Guidance Ontology

---

Algorithm 1: Spec2CDG

Input: *SubRS*: Subsystem Requirement Specifications
Output: *CDG*: Control Dependence Graph of *SubRS*
Data: StructSpec: list of Structured Specifications;
ref: requirement reference;
Declare:
detectKeywords(*S*): detects keywords from KeywordBase in sentence *S*
detectPattern(*S*): detects pattern of sentence *S* from PatternBase
addNode(*CDG*,*X*): creates *CDG* node noted as *X*
addEdge(*CDG*,N_1,N_2): creates edge from node N_1 to node N_2 of *CDG*
Begin
while *not at end of SubRS* **do**
  read current;
  if *understand* **then**
    detectKeywords(current) ;
    if *Keywords detected* **then**
      | detectPattern(current);
    else
      | go back to the beginning of current section;
    if *no pattern detected* **then**
      | create new pattern;
    save structured specification;
  else
    | try to understand

for *element of StructSpec* **do**
  get *ref* from element;
  if *element does not contain ref* **then**
    | determineDependencies(element,StructSpec);
  addNode(*CDG*,element);
  set currentNode element;
  if *element contains many alternatives* **then**
    for *condition of alternatives* **do**
      | addNode(*CDG*,condition);
      | addEdge(*CDG*,currentNode,lastNode);
    set currentNode lastNode;
  if *element contains extern condition* **then**
    get *extcond*;
    addNode(*CDG*,*extcond*);
    addEdge(*CDG*,currentNode,lastNode);
    set currentNode lastNode;
  get *ref*;
  if *ref* **then**
    | addNode(*CDG*,*ref*);
    | addEdge(*CDG*,currentNode,lastNode);
  else
    | ask help;

End

Algorithm 2: creationPattern Procedure

Input: *S*: Sentence, *Ont*: Light Ontology
Output: *P*:A pattern of Sentence *S*
Data: *currentConcept*: A current anlysed concept
Concepts: Detected concepts list of *S*
Inst: Detected instances list of Concepts
Declare:
detectConcepts(*Ont*,*S*): Detect concepts from *Ont* in sentence *S*
detectPossiblePatterns(*S*): Detect possible patterns for sentence *S* from PatternBase based on
detected *keywords*
Begin
init Concepts;
init *Inst*;
init *currentConcept*;
detectPossiblePatterns(*S*);
set Concepts by possible concepts;
while *not at the end of Concepts* **do**
    read *currentConcept*;
    detectInstance(*Ont*,*currentConcept*);

if *Inst is not empty* **then**
    match *P* with Possible Patterns;
    if *P* **then**
        return *P*;

return Nil;

Algorithm 3: detectPattern Procedure

Input: *S*: Sentence, *Ont*: Light Ontology
Output: *P*:A pattern of Sentence *S*
Data: *currentConcept*: A current anlysed concept
Concepts: Detected concepts list of *S*
InstStack: Detected instances list of Concepts
Declare:
detectConcepts(*Ont*,*S*): Detect concepts from *Ont* in sentence *S*
detectPossiblePatterns(*S*): Detect possible patterns for sentence *S* from PatternBase based on
detected *keywords*
Begin
init Concepts;
init *InstStack*;
init *currentConcept*;
detectPossiblePatterns(*S*);
set Concepts by possible concepts;
while *not at the end of Concepts* **do**
    read *currentConcept*;
    detectInstance(*Ont*,*currentConcept*);

if *InstStack is not empty* **then**
    match *P* with Possible Patterns;
    if *P* **then**
        return *P*;

return Nil;

When the algorithm encounters external conditions in the statement, it creates the `extcond` node as well as the statement node and constructs control flow edges between them. An external condition is a conditional statement that refers to a $<Subsystem>$ which is not belong to any of current subsystem stakeholders, unlike to internal condition, which does concern subsystem stakeholders. The definition of [Hull et al.(2004)], a stakeholder is "*an individual, group of people, organisation or other entity that has a direct or indirect interest or stake in a system*". To identify stakeholders of the *SubRS*, we need glossary entries.

If the algorithm encounters a conditional statement *CS* with many alternatives, it creates nodes for each one of them. We consider these alternatives as transition states and they have to be represented on a dependency graph in order to achieve the next state. Each state must to be satisfied.

If there is no specification reference detected, `determineDependencies` is called. Based on similarity computing of different elements of `StructSpec`, the function returns specification references. This function makes use of various techniques of Natural Language Processing as syntactic and semantic processing. After obtaining the pattern and data, a dependence relationship must to be extracted from the specification. It is a complicated phase that needs rules and knowledge to be achieved. First of all, we have to define the term *"dependent on"* and *"directly dependent on"*.

Definition 1. A specification S_2 is *dependent on* specification S_1 (written $S_1 \delta S_2$) if and only if:

- S_1 precedes S_2 in execution
- Execution of S_1 implies execution of S_2 in the future

Definition 2. A specification S_2 is *directly dependent on* specification S_1 (written $S_1 \delta^d S_2$) if and only if:

- S_1 precedes S_2 in execution
- Execution of S_1 implies execution of S_2 in next step

When a specification contains "transfer" keywords such as (*go to, leads to*, ...), it is possible to identify the dependence with other statements. Specifications without "transfer" keywords are difficult to connect with other specifications. If the latter contains a conditional statement, this one could be an internal or external one. If it is an external condition, it will be translated as a node on dependency graph. The latter is directly dependent on current specification.

For all specifications which do not contain any condition, a semantic analysis is required. These specifications may be final states, initial states or statements which include implicit conditions. The elicitation and extraction of implicit data in general are quite a difficult operation, because there is a need for knowledge, rules and techniques.

To determine dependencies between all elements `StructSpec`, we propose some rules such as proposition 1 to facilitate the detection of dependencies. We

assume that *DS* represents Declarative Statements. DS_τ are Declarative Statements with "*Transfer*" and $DS_{!\tau}$ are Declarative Statements without "*Transfer*" keyword. *CS* are Conditional Statements. $DS_\tau \cup DS_{!\tau} \subseteq DS$. A conditional statement is directly dependent on declarative statement with a condition which is expressed in 1.

Proposition 1. *If S_i is a declarative statement without "Transfer" keyword, it exists conditional statement S_j that is directly dependent on S_i.*
$$S_i \in DS_{!\tau} \Rightarrow \exists S_j \in CS\{S_j \delta^d S_i\}$$

Algorithm 4: determineDependencies Procedure

Input: *S*: Structured specification, *StrcutSpec*: List of structured specifications
Output: *Dep*:List of dependencies between *S* and elements of *StructSpec*
Data: *P*: A pattern
D: Data of a structured specification *S*
s: similarity percent
ε: a constant which means the threshold of acceptance of difference between two expressions
Declare:
similarity(e_1, e_2): Similarity percent between two elements e_1 and e_2.
Begin
$P \leftarrow$ getPattern(*S*);
$D \leftarrow$ getData(*S*);
get *objs*;
while *not at the end of objs* **do**
    get *currentobj*;
    for *element of StructSpec* **do**
        if *element is CS* **then**
            $s \leftarrow$ similarity(*currentobj*, element);
            if $s \geq \varepsilon$ **then**
                $D \leftarrow$ element;

We consider that a specification may be expressed as specification $\equiv$ Pattern + Data. In intuitionistic type theory, every term is annotated by its type, only well-typed terms are well-formed [Martin-Lf, Per: 84].

The previous example of structuring specification shows how to express by keywords, concepts and data a sentence. The pattern provides a syntactic description which contains predefined keywords and concepts. The list `Data` provides instances of concepts hence the importance of guidance ontology to determine hierarchical relationships between concepts and instances. **Although** or {**if, when** } keywords introduces a part of specification PRECONDITION. The POSTCONDITION is captured after **shall** or {*to be, to have*} keywords. The semantic of specification are captured by the structure itself. For example, when $< function >$ has as instance "Inform", we expect two objects for the statement: the receiever and the message. The latter may be a POSTCONDITION of the current specification. **has** or **is** in a condition could introduce states. The type of the next concept of statement could be determined by

a dependent function types.

The algorithm 1 for constructing the dependency graph uses text as input and produces the control dependent Graph CDG. The complexity of algorithm 1 is $O(n^3)$ with n being a cost of unit operation.

5 From Flowchart to Control Dependency Graph

Algorithm 5: FC2CDG

Input: *FC*: Flowchart for Subsytem Requirement Specifications *SubRS* with root S_0
Output: *CDG*: Control dependence graph for *P*
Declare:
addNode(*CDG*,*X*): create *CDG* node *X*
addEdge(*CDG*,N_1,N_2): create edge from node N_1 to node N_2 of *CDG*
Begin
init *FC*.currentRegion;
while *more FC nodes* **do**
    get *FC*.currentNode;
    if *FC.currentNode.type is S_0* **then**
        create *FC*.currentNode as *CGD*.root ;
    else if *FC.currentNode.type is S_f or End* **then**
        create *FC*.currentNode as *CGD*.exitNode ;
        exit;
    else
        addNode(*CDG*,*FC*.currentNode);
        set *CDG*.currentNode ;
    if *adjacent(FC,FC.currentNode,FC.currentRegion)* **then**
        get edge between *FC*.currentRegion and *FC*.currentNode;
        get *CDG*.currentRegion
        if *edge.label is Evenment* **then**
            addNode(*CDG*,*FC*.edge);
        else if *edge.label is exitDecision* **then**
            addNode(*CDG*,*FC*.currentRegion $\cup$*FC*.edge);
        addEdge(*CDG*, *CDG*.currentRegion,*CDG*.lastNode);
        addEdge(*CDG*, *CDG*.lastNode,*CDG*.currentNode);
    set currentRegion ;
End

When a flowchart contains more unprocessed nodes, we identify the node and apply rules according to its type. If the node is a root of the flowchart, we create a root of the control dependency graph, or else if it is an "End" node we create an exit node or a simple node is created and it is connected directly to the current region (previous processed node).

Our algorithm 5 for constructing the Dependency Graph takes Flowchart FC and produces the control dependent Graph CDG. For the sake of presentation, we assume that Flowchart is represented with the following types of statements: state or status, action or activity, decision or test, event or external condition.

The time and space complexities of algorithm 5 are $O(n^2)$ and $O(n^2)$ with n being a cost of unit operation.

6 Conclusion

In this paper, we describe an approach for structuring requirements and the creation of a lignt specific domain ontology. The methodology consists of using knowledge from the guidance ontology to create a more specific models. We have therefore proposed an ontology for safety requirements and we have defined the concepts and their hierarchy as well as the relationships between concepts. Our ontology is derived from our understanding of the processes of requirements engineering, as well as its application in the context of its conceptualization. This approach is the first step of a comprehensive methodology to design, verify and validate the model requirements. It is therefore expected in the future works to define rules of inference "Inference rules" and rules of correspondence "Mapping rules" for the formal verification of the specification document. Metrics for the validation and evaluation requirements will be studied and presented in future work.

References

[IEEE (1998)] *IEEE-STD-1220*, 1998;

[Happel et al.(2006)] Happel H. & Seedorf S. Applications of ontologies in software engineering. *In 2nd International Workshop on semantic Web Enabled Software engineering*. 2006;

[Dameron(2005)] Dameron O. Keeping modular and platform independent software up-todate, Benets from the semantic web. *In 8th International Protg Conference*.2005;

[Dieng et al.(2001)] Dieng K. R., Corby O., Gandon F., Giboin A. Golebiowska J., Matta N. & Ribire M. Informatiques Srie Systmes d'information, chapitre Mthodes et Outils Pour la gestion des Connaissances : Une approche pluridisciplinaire du Knowledge Management. *Dunod Edition*2001;

[Sowa(1999)] Sowa J. F. Relating templates to language and logic. *The 21st International Conference on Software Engineering*.1999;

[Kaiya et al.(2005)] Kaiya M. & Saeki H. Ontology based requirements analysis, lightweight semantic processing approach. *In Fifth International Conference of Quality Software (QSIC 2005;)*.

[OMG(2009)] OMG. Ontology Definition Metamodel. Version 1.0, Document Number: formal/2009-05-01

[Hull et al.(2004)] Elizabeth Hull, Ken JAckson and Jeremy Dick. Requirements Engineering. *In Spriger* pages 7, 2004;