Mapping MOF-based requirements representations to ontologies for software reuse

Katharina Wolter[1], Michał Śmiałek[2], Lothar Hotz[1], Stephanie Knab[1], Jacek Bojarski[2], and Wiktor Nowakowski[2]

[1] HITeC e.V., University of Hamburg, Germany
(kwolter,hotz,knab)@informatik.uni-hamburg.de
[2] Warsaw University of Technology, Poland
(smialek,bojarsj1,nowakoww)@iem.pw.edu.pl

Abstract. In this paper, we combine MOF-based software representations and description logic-based mechanisms for facilitating software reuse. All software representations (requirements specifications, design models, code) arising from one project are combined in a *software case* and stored in a repository for later retrieval. For reuse purposes, we use requirements as search indexes. We map metamodel-based requirements specifications to an ontology and use a Description Logic reasoner for classification. This makes implicit taxonomical relations explicit. The inferred taxonomical hierarchy is then used to compute the taxonomical similarity between the current (initial) requirements and those in the repository. Doing so, we retrieve software cases with a high reuse potential. This approach has already been validated in an industrial context.

1 Introduction

The underlying goal of our work is to enable reuse of model-based software development artifacts. For this task, we combine all artifacts that arise in the course of a software development project in a "software case". A *software case* is a set of interlinked models and code written in textual and graphical languages that have their syntax defined formally (normally with a metamodel or a context-free grammar). The contents of a software case are manifested through the contained requirements model, which is specified using a precise and semantic-oriented requirements language. In this language, meaning of words is defined by linking them to WordNet [1]. The requirements model is mapped to other models (using model-based transformations) so that all syntactic elements of these models have their source in at least one element of the requirements model. All software cases that have been developed are stored in a repository for later reuse.

The retrieval of similar software cases from the repository is a key prerequisite for this reuse approach. In order to retrieve such similar software cases we compare requirements models. This is based on the assumption that cases with similar requirements models can have similar design models, too. The retrieval uses a combination of different similarity measures [2], [3]. In this paper, we focus on the ontology-based similarity measure for software cases.

Such an ontology-based similarity measure requires the representation of all software cases (more precisely their requirements models) in an ontology. One resulting advantage of representing them in an ontology is the availability of reasoning techniques provided by Description Logic (DL) reasoners like Pellet or Racer[3]. One of these reasoning techniques (namely classification) supports our similarity measure. The classification makes implicit relations between elements in the ontology explicit by changing the taxonomical hierarchy. These newly inferred relations can then be used by our ontology-based similarity measure.

The rest of this paper is structured as follows: Section 2 sketches the facilities Description Logics supply and elaborates on main questions that have to be answered when Description Logics are applied in a certain domain. In Section 3, we introduce the general reuse approach in which our ontology-based similarity measure is embedded and we describe the MOF-based requirements specification language (RSL). In order to realize the similarity measure we had to construct an ontology containing all requirements models, the related metamodel, and the linked elements from WordNet (see Section 4 for details). The use of classification as a reasoning technique for supporting software retrieval is explained in Section 5. Our ontology-based similarity measure uses this inferred taxonomic hierarchy (Section 6). Section 7 describes the validation experiments as well as first results. Finally, the approach is discussed (Section 8) and a summary concludes the paper.

2 DL Modeling and Reasoning Facilities

For our current task (ontology-based similarity measure), we need to select from a diverse range of modeling and reasoning facilities of Description Logic. It provides facilities to define *concepts* and *roles* (or sometimes called *properties*) between two concepts. A concept (given with a unique name) indicates the set of *individuals* that belongs to it, and a role indicates a relationship between concepts. Concepts and roles are combined via constructors to so-called *descriptions*. Typical constructors are those of $\mathcal{ALCHIF}$, i.e. Attributive Language with universal restrictions, existential qualification, concept intersection ($\mathcal{ALC}$), role hierarchy ($\mathcal{H}$), inverse roles ($\mathcal{I}$), and functional properties ($\mathcal{F}$). With these constructors, one can define a specialization relation between concepts, which provides *superconcepts* and *subconcepts* in a taxonomy, i.e. *general* or *specific* concepts. Furthermore, a Description Logic allows to distinguish between *primitive* (i.e. necessary) and *defined* (i.e. necessary and sufficient) concepts, indicated with `implies` and `equivalent`, respectively. If several somehow combined roles are sufficient for a concept, an individual who fulfills such roles would be part of the set the concept describes. If an instance belongs to a concept, the instance fulfills the necessary roles of that concepts.

Given such modeling facilities, DL reasoners provide reasoning techniques. Examples are (see `www.w3.org/Submission/owl11-tractable/` or [4]):

– Concept Subsumption (classification): A concept A subsumes B, when the description of A is more general than the description of B.

[3] `clarkparsia.com/pellet`, `www.racer-systems.com`

- Instance Checking: Given an individual a and a concept A, verify whether a is an instance of A.
- Ontology Consistency: Check whether a given ontology is not contradictory.
- Concept Satisfiability: Given a concept A, verify whether the description of A yields a non-empty set of individuals.
- Conjunctive Query Answering: Given an ontology O and a conjunctive query q, return the answers of the query with respect to O.

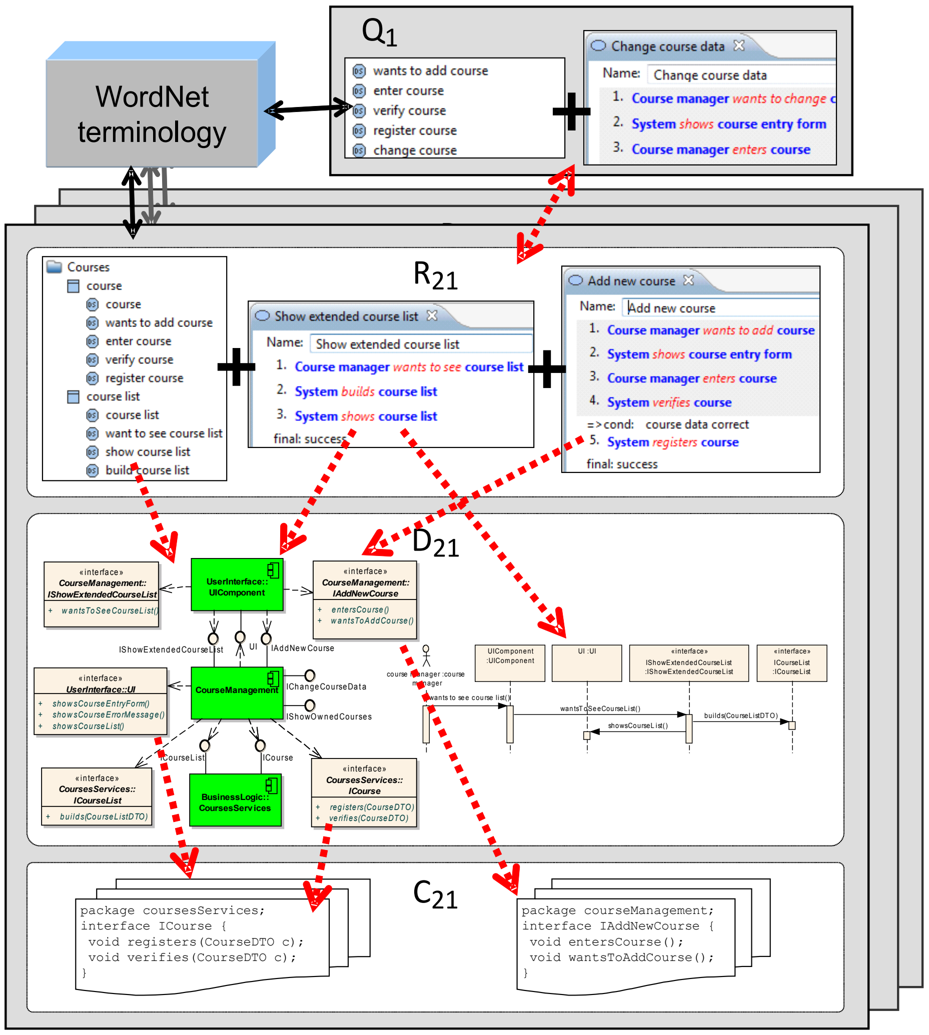
Generally, when ontologies should effectively be applied in a task, like here in ontology-based similarity computation, four design decisions are concerned:

1. What reasoning techniques should be used for supporting the solution of the task? In our case, classification is used (see Section 5 for details).
2. What entities should be mapped to the ontology? In our case, what elements of a software case are relevant for realizing the reasoning technique? Section 3 introduces the main elements of a software case and Section 4 explains the mapping to the ontology.
3. What concepts should be modeled as defined concepts and which should be necessary? Especially defined concepts enable reasoning tasks (Section 4).
4. Which logical language should be used for solving the task (see Section 4 for details)? This comes from previous decisions which impact the expressiveness of the language and computational efficiency.

3 Model-driven Software Cases

Figure 1 shows an example of a software case. Elements of the requirements model within the software case (R) are interlinked with appropriate elements of the design model (D), which, in turn, are reflected in code (C). Design and code form a solution to the software problem manifested by the requirements. The initial requirements for a new problem (Q) can be compared with the requirements models of already existing software cases. Thus, the requirement models serve as indexes to the design models which facilitates searching and retrieval of past solutions that can be applied to the current problem. For the indexing mechanism to work properly we have designed a special-purpose Requirements Specification Language (RSL). Its syntax contains detailed constructs which allow for automatic comparison and transformation to the design models.

RSL enables two scenarios of software development: constructing and storing a software case, and retrieving a software case. When starting a new software project, initially we create its requirements model (see R_{21} in Figure 1). The model contains the description of the application logic with a separate description of the problem domain. Such a complete requirements model written in controlled language, is automatically transformed into the design model (see D_{21}). This model can be manually updated to conform to certain non-functional constraints. Then, the code (see C_{21}) is generated, which includes also some of the contents of the methods (the application logic). Finally, all these artifacts, together with the mappings generated between them are stored in a software case repository.


WordNet terminology
Q1
wants to add course
enter course
verify course
register course
change course
Change course data
Name: Change course data
1. Course manager wants to change c
2. System shows course entry form
3. Course manager enters course
R21
Courses
course
course
wants to add course
enter course
verify course
register course
course list
course list
want to see course list
show course list
build course list
Show extended course list
Name: Show extended course list
1. Course manager wants to see course list
2. System builds course list
3. System shows course list
final: success
Add new course
Name: Add new course
1. Course manager wants to add course
2. System shows course entry form
3. Course manager enters course
4. System verifies course
=>cond: course data correct
5. System registers course
final: success
D21
«interface» CourseManagement::IShowExtendedCourseList
+ wantsToSeeCourseList()
UserInterface::UIComponent
«interface» CourseManagement::IAddNewCourse
+ entersCourse()
+ wantsToAddCourse()
IShowExtendedCourseList
UI
IAddNewCourse
«interface» UserInterface::UI
+ showsCourseEntryForm()
+ showsCourseErrorMessage()
+ showsCourseList()
CourseManagement
IChangeCourseData
IShowOwnedCourses
ICourseList
ICourse
«interface» CoursesServices::ICourseList
+ builds(CourseListDTO)
BusinessLogic::CoursesServices
«interface» CoursesServices::ICourse
+ registers(CourseDTO)
+ verifies(CourseDTO)
course manager :course manager
UIComponent :UIComponent
UI :UI
«interface» IShowExtendedCourseList :IShowExtendedCourseList
«interface» ICourseList :ICourseList
wants to see course list()
wantsToSeeCourseList()
builds(CourseListDTO)
showsCourseList()
C21

```
package coursesServices;
interface ICourse {
 void registers(CourseDTO c);
 void verifies(CourseDTO c);
}
```

```
package courseManagement;
interface IAddNewCourse {
 void entersCourse();
 void wantsToAddCourse();
}
```

Fig. 1. Example of an elementary software case (R_{21},D_{21},C_{21}) with a query (Q_1)

After some time, when we start a next project, we write an initial requirements model which is used to build a query (see Q_1) to search for past solutions that can be reused. The query engine compares the query with the requirements models of past software cases. The most similar software cases or their parts are picked by the team and copied to the current workspace for merging with the current project. After merging, the newly created software case can be stored in the repository for further reuse.

Figure 1 explains the details of RSL. It shows three use cases (see boxes with tabs within Q_1 and R_{21}) which are the basic units of functionality in RSL. Use cases have their detailed representations in the form of textual scenarios (graphical versions also can be used). Every such scenario is a sequence of numbered

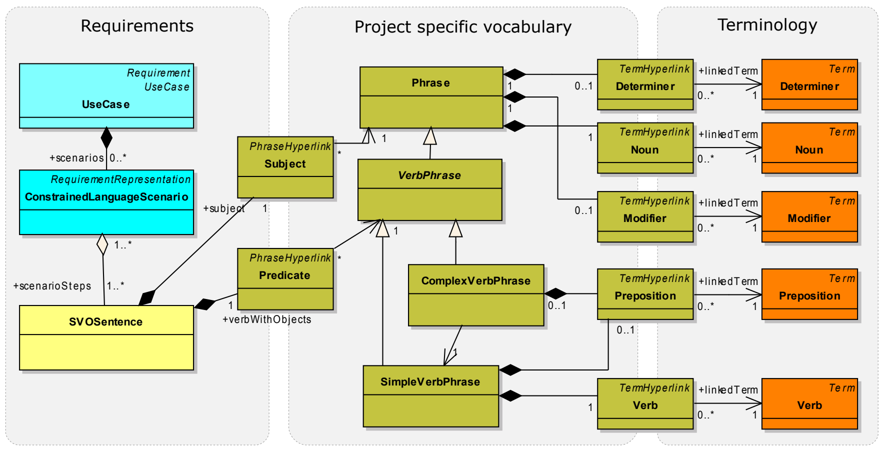


Fig. 2. Fragment of the RSL metamodel

sentences in the SVO grammar – consisting of a subject (S), a verb (V) and one or two objects (O), e.g. *System shows course list*. Actually, scenario sentences are not pure text but they are built of hyperlinks pointing to notions in a domain vocabulary (e.g. *course list*). These notions can have their definitions specific to the system's problem domain. Notions can also contain domain statements which are verb phrases associated with the given notion (e.g. *show course list*). An example domain vocabulary is shown in Figure 1 to the left of the scenarios. In order to make requirements models comparable for reuse purposes, we have to additionally link all elements of the domain vocabulary to one global terminology, like WordNet [1] that contains meanings of words in the general context. This makes possible to calculate semantic-oriented similarity of different requirements models as it will be shown in the following sections. In this context it can be noted that RSL models can both form parts of software cases (R_{21}) and serve as queries (Q_1) for software case retrieval.

To facilitate automatic transformations, the RSL syntax is defined through a meta-model in MOF. This was a natural choice which facilitates constructing transformations to design models written in the MOF-based UML. It also allows for mapping RSL to an ontology. For transformations we use the MOLA transformation language (see [5]).

A relevant fragment of the RSL metamodel expressed in MOF is presented in Figure 2. The metamodel is composed of three parts, divided with metaclasses constituting hyperlink relations. First part is the requirements part (left frame). In RSL, requirements can be modeled as general requirements written in natural language (not included in Figure 2) or as use cases with scenarios defined in constrained language. Scenarios (ConstrainedLanguageScenario[4] metaclass) are composed of SVOSentences. Every such sentence is composed of a Subject and a Predicate which are in fact hyperlinks to definitions of the proper phrases in a

[4] With this font we will denote elements of the RSL metamodel.

vocabulary (center frame). The Subject points to a Phrase containing one Noun and optionally one Modifier and one Determiner. The Predicate points to one of the two VerbPhrase subclasses. It contains the VO(O) part of the sentence. If the sentence contains only the direct object (SVO sentence), the predicate points to a SimpleVerbPhrase. This subtype of VerbPhrase contains a Noun (inherited from the Phrase) and an additional Verb added before the Noun (eg. ,,add user"). Phrases containing both the direct and indirect object (VOO) are stored in a ComplexVerbPhrase. This subtype of the VerbPhrase links an instance of a SimpleVerbPhrase (the VO part of the phrase) followed by a Preposition and an indirect object (the Noun inherited from the Phrase, eg. ,,add user to user list"). In RSL, the vocabulary is project specific and contains notions' and phrases' definitions characteristic to the current problem domain. All the words used in the project-specific phrases are summarized in a terminology (see the right frame). The TermHyperlinks connect the Phrases and the Terms. Each Term links a WordNet element (by referring to its ID) in order to define the terms's meaning in a general context. WordNet is a sematic lexicon which currently consists of more than 200.000 synonyms grouped in more than 100.000 synonym sets (called synsets, see [4]). Semantic relations (like is-a and has-parts) connect these synsets.

This three-level language infrastructure enables clear separation between the behavior specification (use case scenarios' steps) and the description of the domain (vocabulary). The third level – the terminology – defines the meaning of words in a global context and thus gives the possibility to perform meaningful comparison of different requirements models.

For expressing design models within software cases, we use a subset of UML elements defined through a profile. A design model is generated automatically as a result of a transformation that takes a requirements model in RSL as its source. The transformation procedure adds to the information contained in requirements all the details of the technical solution: the architectural style, design patterns, etc. In the example we present the results of a transformation from requirements R_{21} to design D_{21}. The example uses a 3-tier architectural model as the transformation target. Use cases are transformed into interfaces of the application logic layer and domain element are transformed into interfaces of the business logic layer and also into data access objects. Sequences of scenario sentences can be transformed into sequence diagrams on the architectural level. For more details on the transformation rules see [6]. During the transformation, the mapping links between elements of the requirements model and the design model are created. By following these links, the design elements and the resulting code (see C_{21}) can be retrieved for reuse.

Validation efforts conducted in the industrial context proved that despite of its very detailed meta-model, the language is readable and understandable for its users (see [7]). The full specification of the RSL can be found in [8].

4 Mapping the MOF-based Models to an Ontology

In order to use the classification facilities provided by DL reasoners we had to represent the RSL metamodel, the requirements models and the linked WordNet elements in an ontology using ontology languages like OWL [9] or KRSS [10]. The goal was to represent the requirements models in a way such that similar requirements have short taxonomical distances after classification. This representation is explained in the following, starting with the RSL metamodel.

4.1 Mapping the RSL Metamodel

The RSL metamodel consists of 127 classes, sub- and superclasses, and many associations between them. These classes and associations are represented in the ontology as follows: classes of the metamodel are represented with concepts, generalization relations are represented with specialization relations, and associations are represented with roles. Furthermore, the associations of a class in the metamodel *define* the class, i.e. if an object with such relations exists, then it belongs to that class and if an object belongs to a class it has the relations of that class (see Figure 3). The classes of the RSL metamodel are furthermore disjoint, because every element of a requirements model is an instance of exactly one class of the metamodel.

Although navigation across associations is bidirectional in the RSL metamodel, we do not use inverse roles in the ontology since the similarity computation only requires roles *directing down* from the concept representing the requirements model to the synsets of WordNet. Likewise, we avoided a role hierarchy because only a relatively flat hierarchy of associations is given in the RSL metamodel. Finally, functional properties are not needed for defining concepts because they relate a class to a primitive type not to another concept. Thus, we can use the DL language $\mathcal{ALC}$ instead of the more complex language $\mathcal{ALCHIF}$.

4.2 Mapping Requirements Models

The requirements models of specific software cases are represented through instances of classes of the RSL metamodel. This is illustrated in Figure 4. The RSL metamodel can be found on M2 while the specific requirements models are located on M1 (in the context of the MDA metamodeling layers). Thus, the SVO sentences defined in a particular requirements model are *instances* of the class SVOSentence in the RSL metamodel. Please note that we represent these instances with further *concepts* in the ontology. They are mapped to subconcepts of the concepts representing the classes of the RSL metamodel. This is

```
(equivalent SVOSentence
   (and ConstrainedLanguageSentence
        (some subject Subject)
        (some predicate Predicate)))
```

Fig. 3. Definition of an upper model concept representing the RSL class SVOSentence.

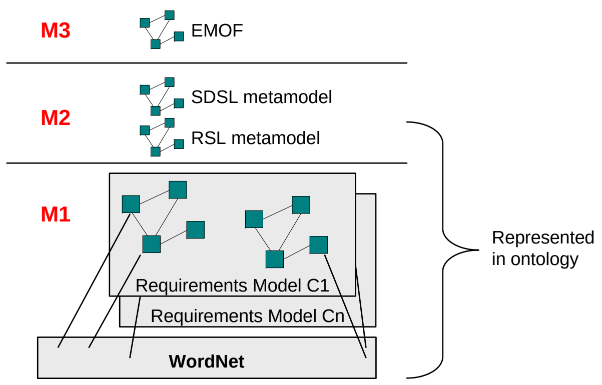


Fig. 4. Requirements models and their metamodel are represented in the ontology.

illustrated in Figure 5 which shows a part of an ontology, which we created manually for illustration purposes. Concepts for SVO sentences are coded by a software case id starting with C and a readable string reflecting the text of the sentence (e.g. `SVOSentenceC2GuestChoosesEquipment`[5]). The concept `SVOSentence` is the superconcept of four SVO sentences contained in distinct requirements models.

As a result, the concepts representing the classes of the RSL metamodel form the upper part of our ontology. Thus, the part of the ontology representing the model from the M2 level is called *upper model* of the ontology. The concepts representing the elements of specific requirements models from the M1 level are modeled as specializations of these concepts and form the lower part of our ontology.

We do not use *individuals* for representing the elements of particular specifications because taxonomical relationships cannot be computed between individuals. In our approach this is the key to infer taxonomical relationships between the elements of requirements models, because the similarity is based on taxonomical relations.

Thus, each class of a specific requirements model is mapped to a concept with a unique id as its name and the corresponding upper model concept as its superconcept. The concepts representing the elements of requirements models

[5] With this font we will denote elements defined in the ontology.

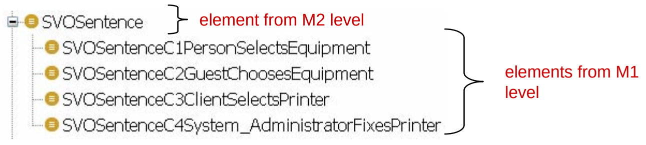

SVOSentence
SVOSentenceC1PersonSelectsEquipment
SVOSentenceC2GuestChoosesEquipment
SVOSentenceC3ClientSelectsPrinter
SVOSentenceC4System_AdministratorFixesPrinter

Fig. 5. Concepts in the ontology representing classes from the M2 and the M1 level.

```
(equivalent SVOSentenceC2GuestChoosesEquipment
   (and SVOSentence
        (some subject SubjectC2Guest)
        (some predicate PredicateC2ChoosesEquipment)))
(equivalent PredicateC2ChoosesEquipment
   (and Predicate
       (some verbPhrase SimpleVerbPhraseC2Choose)))
(equivalent SimpleVerbPhraseC2Choose
   (and SimpleVerbPhrase
        (some verb PhraseVerbLinkC2Choose)
        (some object NounPhraseC2Equipment)))
(equivalent PhraseVerbLinkC2Choose
   (and PhraseVerbLink
        (some linkedVerb VerbC2Choose)))
(equivalent VerbC2Choose
   (and Verb
        (some termLinksToWordnetEntry ChooseSelectSynset)))
(equivalent NounPhraseC2Equipment
   (and NounPhrase
        (some noun NounLinkC2Equipment)))
(equivalent NounLinkC2Equipment
   (and NounLink
        (some linkedNoun NounC2Equipment)))
(equivalent NounC2Equipment
   (and Noun
        (some termLinksToWordnetEntry EquipmentSynset)))
```

Fig. 6. Parts of the definition of the SVO sentence "Guest chooses equipment.".

are related through roles defined in the upper model. Furthermore, these roles *define* the requirements model concepts (see Fig. 6). Thus, not only necessary relations (i.e. subclass) but necessary and sufficient relations are used for relating software cases. This modeling enables the reasoning techniques described in the Section 5.

This general mapping can be used to convert all elements of a requirements model to our ontology. However, this is not necessary since RSL requirements models contain some elements not relevant for our ontology-based similarity measure like all TermHyperlinks in Figure 2 (see [11] for details). Mapping these elements would cost significant amount of computing time and would not improve the similarity measure. Thus, we only map RSL elements that are relevant for our similarity calculation.

4.3 Mapping WordNet Elements

The WordNet metamodel consists of several classes like Synset, Wordform, Gloss etc. NounSynsets are related via the relation hyponym and hypernym in a taxonomy. Classification only requires this taxonomical relation between synsets and the fact that synsets are distinct. Thus, we only map synsets from WordNet. They are represented with concepts and the synset taxonomy is represented with the subconcept relation.

The role `termLinksToWordNetEntry` relates each term of a requirements model to a synset (see Fig. 6). However, it is not necessary to map all synsets of WordNet (i.e. almost 117.000) into the ontology. For our purposes, it is only necessary to map the synsets that are linked by a term and all the predecessors of these synsets in the synset taxonomy. Synsets have only taxonomical relations between

```
(implies ChooseSelectSynset VerbSynset) (implies FixSynset VerbSynset)
(implies PersonSynset NounSynset)
(implies ComputerUserSynset PersonSynset)
(implies System_AdministratorSynset ComputerUserSynset)
(implies UserSynset PersonSynset) (implies ConsumerSynset UserSynset)
(implies CustomerClientSynset ConsumerSynset)
(implies GuestSynset CustomerClientSynset)
(disjoint ComputerUserSynset UserSynset)
(disjoint ChooseSelectSynset FixSynset)
```

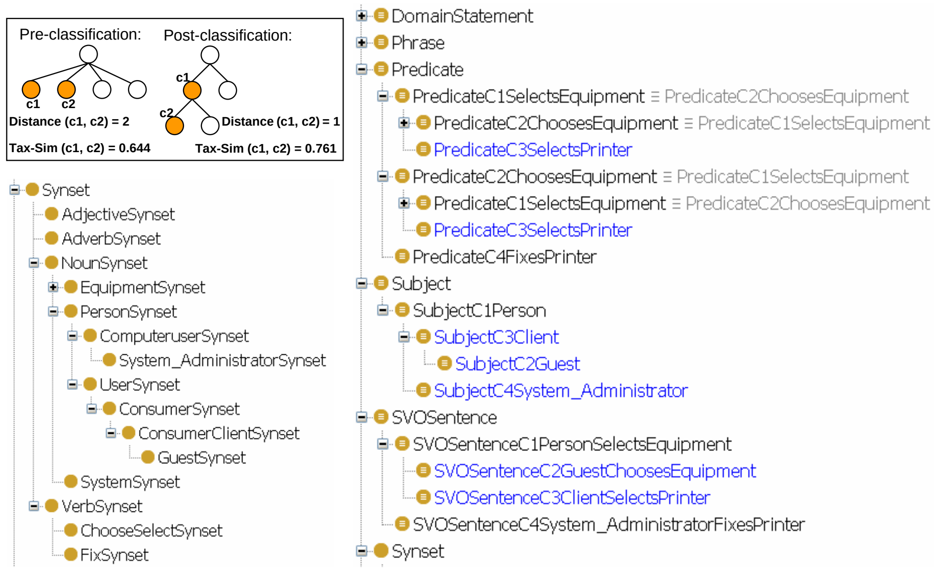
Fig. 7. Mapping synsets into concepts.

themselves (i.e. no further roles). Thus, the taxonomical relations are necessary conditions, i.e. implies is used for describing them. Furthermore, synsets that are siblings are disjoint to each other (see Fig. 7).

5 Classifying the Ontology

In this section, we show what can be gained by classifying the constructed ontology. Using the mapping described in the previous section we can represent requirements models with concepts and roles in an ontology. However, the requirements models do not contain information that directly relates the diverse concepts of distinct requirements models. As a result, all concepts are direct subconcepts of upper model concepts. In Figure 5 e.g., all SVO sentences are direct subconcepts of the upper model concept `SVOSentence`. In this initial ontology, only the synset taxonomy provides a hierarchical structure (see Fig. 8, left side).

Figure 8 shows the SVO sentences from Figure 5 plus some additional concepts but after classification. The synset taxonomy provided by WordNet and the fact that the



```
Synset
    AdjectiveSynset
    AdverbSynset
    NounSynset
        EquipmentSynset
        PersonSynset
            ComputeruserSynset
                System_AdministratorSynset
            UserSynset
                ConsumerSynset
                    ConsumerClientSynset
                        GuestSynset
        SystemSynset
    VerbSynset
        ChooseSelectSynset
        FixSynset
```

```
DomainStatement
Phrase
Predicate
    PredicateC1SelectsEquipment ≡ PredicateC2ChoosesEquipment
        PredicateC2ChoosesEquipment ≡ PredicateC1SelectsEquipment
        PredicateC3SelectsPrinter
    PredicateC2ChoosesEquipment ≡ PredicateC1SelectsEquipment
        PredicateC1SelectsEquipment ≡ PredicateC2ChoosesEquipment
        PredicateC3SelectsPrinter
    PredicateC4FixesPrinter
Subject
    SubjectC1Person
        SubjectC3Client
            SubjectC2Guest
        SubjectC4System_Administrator
SVOSentence
    SVOSentenceC1PersonSelectsEquipment
        SVOSentenceC2GuestChoosesEquipment
        SVOSentenceC3ClientSelectsPrinter
    SVOSentenceC4System_AdministratorFixesPrinter
Synset
```

Fig. 8. Left side: synset taxonomy mapped from WordNet. Right side: cutout of inferred hierarchy: classification results are emphasized in gray and blue font.

concepts representing a requirements model are defined concepts and thus, strongly related between each other, allow for the classification described in the following.

Our mapping of RSL metamodel and requirements models enables a DL reasoner to use the synset taxonomy in order to classify the concepts of the requirements models, i.e. to compute, which concepts are equivalent (Fig. 8, upper part in gray marked by $\equiv$) or, which concepts can be taxonomically structured (Fig. 8, lower part in blue). Relations of the synset taxonomy are propagated one by one to the subjects of requirements models of $C1$, $C2$, $C3$, and $C4$. Namely, `SubjectC3Client` and `SubjectC4System_Administrator` are subconcepts of the concept `SubjectC1Person` due to the fact that `CustomerClientSynset` and `System_AdministratorSynset` are subconcepts of `PersonSynset`, respectively. Thus, after classifying the ontology, the taxonomy defined in WordNet is also reflected in the concepts representing the specific requirements models. Even more interesting is the impact on structural concepts like `SVOSentence`. In Figure 8 for example, the SVO sentences of case $C2$ and $C3$ are classified as subconcepts of the SVO sentence of case $C1$. This is due to the fact that in both sentences a specific type of person (being either a guest or a client) *chooses* an type of equipment. Since the SVO sentence of $C4$ is about *fixing* a printer instead of selecting one, the sentence is not a subconcept of the SVO sentence of case $C1$. Thus, classifying the ontology makes these taxonomical relations explicit. Please note that Figure 8 shows only parts of this information.

This kind of classification takes the structure, i.e. the roles between the concepts, into account. Through this classification, a different taxonomical distance between the concepts of requirements models is introduced and will lead to different similarities. Before classifying the ontology, the taxonomical distance between all SVO sentences was equal (see Fig. 5). Classification reduces the distance between SVO sentences of $C2$ and $C1$ while it increases the distance between sentences of $C2$ and $C4$ (comp. Fig. 5 and Fig. 8). This effect of classification on distances is sketched in the box in Figure 8. By using the WordNet link and the provided mapping, our similarity measure considers the meaning defined in the requirements models.

6 Similarity Computation

Once the ontology has been constructed and classified we use the inferred taxonomical hierarchy for computing similarity values. For this purpose, a query is compared with each requirements model contained in the repository where the query is a (potentially partial) requirements model. The goal of similarity computation is to get a value between 0 and 1 that indicates the similarity between a query and a software cases; 0 denoting no similarity and 1 denoting that the query is completely contained in the software case.

Please note that our measure differs from most similarity measures. Typically, similarity measures positively correlate with the amount of commonality between query and case and at the same time negatively correlate with the amount of difference (see [12]). The latter however, is not wanted for our similarity measure. A query element that is not found in a software case should of course result in a lower similarity value but if a software case contains elements that are not part of the query, this should not have a negative impact on the similarity value. This is due to the assumption that additional elements in the software case are potential for reuse and thus wanted. Supported by a tool, the user can browse these additional elements and select those that are valuable in the current software development project while others are omitted. In order to allow for this, our measure is asymmetric.

The assumption made here is that the taxonomical distance between two concepts can be the basis of such a similarity measure. Our similarity measure compares pairs of concepts in the classified taxonomy. This comparison includes both the concept placement in the taxonomical hierarchy (*distance-based similarity*) and the roles and role fillers that the concepts define (*role-based similarity*) (see [13] for a similar approach).

For computing the *distance-based similarity* between two concepts c1 and c2, the distance of both concepts to the least common subsumer (LCS) is computed, both values are added and a value describing the distance-based similarity is returned:

```
1. lcs = leastCommonSubsumer(c1, c2)
2. distance1 = pathLength(c1, lcs)
3. distance2 = pathLength(c2, lcs)
4. distance = distance1 + distance2
5. return $\frac{1}{ln(distance+e)}$
```

The algorithm has the following properties: The division ensures values between 0 and 1. The further away the concepts are from each other, the smaller the similarity value gets. The logarithm ensures higher similarity values for low distances, than, for example, $\frac{1}{distance}$ would provide. Finally, the addition of e ensures a similarity value of 1 for distance 0, i.e. if c1 and c2 are in fact the same concept. The box in Figure 8 illustrates the effect of classification on the distance-based similarity measure. By classification the distance between c1 and c2 is reduced from 2 to 1. The associated similarity values of the distance-based measure are also given in the figure.

The basic idea of *role-based similarity* is that the similarity of two concepts depends on the similarity of their subgraphs. When comparing roles, their most important aspects are their fillers: concepts or concrete domains that specify the value of a role. When comparing two concepts, the function of role-based similarity is recursively applied. The recursion terminates when two concepts without roles are compared; their similarity is given by the distance-based similarity function. The roles of both concepts are recursively compared and similarities are summed up for every concept. A value describing role-based similarity is returned: 1 means both concepts have the same roles, and the smaller the value, the less their roles have in common.

Both aspects can be computed independently from each other. Therefore, we defined two algorithms, one computing the distance-based similarity between two concepts and another one comparing the common roles of concepts. The similarity between two concept definitions is the sum of the distance-based similarity and the role-based similarity divided by two.

7 Experiments

During the development of our approach we used small examples for testing and illustration purposes (see previous sections). In order to evaluate the presented similarity measures we conducted two types of experiments. On the one hand, we produced several small software cases by varying specific elements e.g. of a sentence, in order analyze the impact of these modifications. On the other hand, we applied the approach to software cases produced by software development teams in the industrial context (see Acknowledgements).

Technical Setting. The RSL metamodel was implemented as part of a comprehensive tool suite [14]. Internally, the tool uses a graph representation of the metamodel (see

[15]). This representation provides the input for the presented ontology construction. It also serves as the basis for transforming into design and storing of software cases.

A JAVA-based converter maps the RSL metamodel and the requirements models to an OWL ontology by using the mapping described in this paper. For debugging the OWL ontology we use Protégé. Protégé also provides interfaces to the DL reasoners Pellet and Racer. For debugging inference behavior we apply Pellint.[6] The similarity computation is a further JAVA-component developed by us. It traverses the classified ontology for computing similarity values as presented in Section 6. This similarity measure is combined with other measures and integrated in the above mentioned software development tool.

Experiments. Firstly, we created almost 50 small, artificial requirements models. These models vary only in particular elements e.g. one word is added, one word is replaced by a more specific term, one sentence is added etc.. Comparing the similarity values for these requirements models we could analyze the impact of particular modifications in detail. Furthermore, we could show that the ontology-based similarity measure computes the same similarity ranking for these test cases as considered plausible by us beforehand.

For applying the approach in a broader scale, 16 industrial software cases have been created by four software development organizations using the above mentioned tool. The application domains are in the area of internet banking, investment funds management, emergency systems, forestry systems, financial contract systems, and funding systems. The requirements models contained 261 requirements written in RSL in total, which mapped to more than 30.000 defined concepts. We were able to reduce the number of defined concepts to about 8.000 by converting only concepts that are relevant for the ontology-based similarity computation (see Section 4).

In addition, 8 more partial requirements models were created. These models were used to query the repository of the above mentioned cases. The development teams were to determine the compliance of their understanding of similarity and the computed measures. This was subject to UTAUT analysis [16] which included appropriate questionnaires asking about the similarity measures. The results of the analysis showed general positive attitude and acceptance of the developers. It resulted also with several remarks which allow for additional improvements in the classification approach. Namely, further work will extend the number of tractable concepts (which is limited through the capabilities of the used DL-reasoners). The currently gathered repository of software cases gives good means to further experiment in this direction.

8 Discussion and Related Work

As can be seen in Figure 4 the constructed ontology crosses the border between the levels M1 and M2. This is due to the fact that the instance-of relationship between an element on Level M1 (e.g. a specific SVO sentence) and its class (like the RSL class SVOSentence) on Level M2 is mapped to a specialization relation of appropriate concepts in the ontology. Or, as we introduced it in Section 4, the metamodel of RSL is represented as an upper-model in the ontology. By doing so, definitions of the metamodel can be used within the ontology, i.e. by reasoning techniques. This mapping is slightly different from those described in [17] where a level-wise mapping is proposed. With our mapping the detailed metamodel of RSL is used in the ontology for preclassifying the elements of M1, i.e. the elements of a requirements model. Thus, the M1-elements

[6] `protege.stanford.edu`, `pellet.owldl.com/pellint`

are not subconcepts of the root `Thing` but of specific concepts like `SVOSentence` or `VerbPhrase`. The preclassification provokes that the taxonomical distance between different elements is higher than it would have been without preclassification where all elements are subconcepts of `Thing`.

Other reasoning techniques like Instance Checking could be used for verifying if a given requirements model complies to the RSL metamodel (i.e. model confirmation). However, this is not needed in our case, since the software development tool mentioned above already ensures that each requirements model is compliant with the RSL metamodel.

Some approaches allow the user to use natural language (e.g. [18]). This results in the need to parse the text which can lead to more than one possible model due to language ambiguities. On the contrary, formal languages which avoid such ambiguities like Z are not easy to understand for all stakeholders. RSL combines the advantages of both alternatives. It is understandable for its users [7] and provides unambiguous requirements models that can directly be mapped onto ontology elements.

ORE [19] uses a similar approach to define the meaning of words in a specification; they are related to concepts of an ontology. However, OREs support is based on a domain-specific ontology containing domain-dependent inference rules that need to be developed and maintained. The same holds for the ontology-driven requirements analysis described in [20]. In contrast, our approach is independent of domain-specific inference rules. We only require the mature semantic lexicon WordNet as input for our approach.

9 Summary

In this paper, we presented our new combination of MOF-based requirements models and reasoning techniques from Description Logic for supporting the reuse of similar software cases. We showed how we represent the MOF-based models in an ontology. Special in our approach is that we combine elements from different levels (i.e. M2 and M1) within one taxonomy in our ontology. Doing so, we are able to infer taxonomical relations implicitly contained in requirements models. These inferred taxonomical relations are used by our ontology-based similarity measure. Current experiments using industrial software cases show promising validation results.

Acknowledgments. This work is partially funded by the EU: Requirements-Driven Software Development System (ReDSeeDS) (contract no. IST-2006-33596 under 6FP). The project is coordinated by Infovide, Poland with technical lead of Warsaw University of Technology and with University of Koblenz-Landau, Vienna University of Technology, Fraunhofer IESE, University of Latvia, HITeC e.V. c/o University of Hamburg, Heriot-Watt University, PRO DV, Cybersoft and Algoritmu Sistemos.

References

1. Fellbaum, C., ed.: WordNet: An Electronic Lexical Database. MIT Press (1998)
2. Wolter, K., Krebs, T., Hotz, L.: A combined similarity measure for determining similarity of model-based and descriptive requirements. In: Proceeding of the Artificial Intelligence Techniques in Software Engineering Workshop (AISEW) at the ECAI 2008. (2008) 11–15
3. Bildhauer, D., Horn, T., Ebert, J.: Similarity-driven software reuse. In: International Workshop on Comparison and Versioning of Software Models (CVSM 2009). (2009)

4. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press (2003)
5. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. Lecture Notes in Computer Science **3599** (2004) 14–28
6. Kalnins, A., Kalnina, E., Celms, E., Sostaks, A., Schwarz, H., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Kavaldjian, S., Falb, J.: Reusable case transformation rule specification. Project Deliverable D3.3, ReDSeeDS Project (2007) www.redseeds.eu.
7. Mukasa, K.S., Jedlitschka, A., Graf, C., Klöckner, K., Eisenbarth, M., Steinbach-Nordmann, S.: Requirements specification language validation report. Project Deliverable D2.5.1, ReDSeeDS Project (2007)
8. Kaindl, H., Śmiałek, M., Svetinovic, D., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T., Schwarz, H., Bildhauer, D., Brogan, J.P., Mukasa, K.S., Wolter, K., Krebs, T.: Requirements specification language definition. Project Deliverable D2.4.1, ReDSeeDS Project (2007) www.redseeds.eu.
9. W3C: OWL Web Ontology Language Overview. (2004) http://www.w3.org/TR/owl-features.
10. Patel-Schneider, P.F.: Description-logic knowledge representation system specification from the KRSS group of the ARPA knowledge sharing effort. Technical report, DARPA Knowledge Representation System Specification (KRSS) Group of the Knowledge Sharing Initiative (1993)
11. Hotz, L., Wolter, K., S., K., Solth, A.: Ontology-based Similarity of Software Cases. In to appear, ed.: International Conference on Knowledge Engineering and Ontology Development. (2009)
12. Borgida, A., Walsh, Thomas, J., Hirsh, H.: Towards measuring similarity in description logics. In: International Workshop on Description Logics, Edinburgh, Scotland, DL2005 (2005)
13. González-Calero, P.A., Gómez-Albarran, M., Díaz-Agudo, B.: Applying DLs for retrieval in case-based reasoning. In: Proc. of the 1999 International Workshop on Description Logics (DL'99). (1999)
14. Śmiałek, M., Ambroziewicz, A., Bojarski, J., Nowakowski, W., Straszak, T.: ReDSeeDS: Requirements Driven Software Development System. to be presented at ICSR 09 (2009) tool demo.
15. Dahm, P., Widmann, F.: GraLab - Das Graphenlabor. Projektbericht 4.3.0, University of Koblenz-Landau, Institute for Software Technology (2003)
16. Venkatesh, V., Smith, R.H., Morris, M.G., Davis, G.B., Davis, F.D.: User acceptance of information technology: Toward a unified view. MIS Quarterly **27** (2003) 425–478
17. Zhao, Y., Pan, J.Z., Pareiras, F.S., Walter, T., Gröner, G., Ren, Y., Jekjantuk, N.: Successful Stories and Potential Patterns on Applying Ontologies in Software Engineering. Project Deliverable WP3-D6, MOST Project (2008)
18. Kleiner, M., Albert, P., Bezivin, J.: Configuring Models for (Controlled) Languages. In: Proceedings of the IJCAI-09 Workshop on Configuration. (2009)
19. Kaiya, H., Saeki, M.: Using domain ontology as domain knowledge for requirements elicitation. In: Proc. of 14th IEEE International Requirements Engineering Conference (RE'06), IEEE CS (2006) 189–198
20. Liu, W., He, K.Q., Wang, J., Peng, R.: Heavyweight semantic inducement for requirement elicitation and analysis. In: SKG '07: Proceedings of the Third International Conference on Semantics, Knowledge and Grid, Washington, DC, USA, IEEE Computer Society (2007) 206–211