# Requirements Analysis Driven by Ontological and Production Models

Marina Sh. Murtazina
Novosibirsk State Technical University
Novosibirsk, 630073
murtazina@corp.nstu.ru

Tatiana V. Avdeenko
Novosibirsk State Technical University
Novosibirsk, 630073
avdeenko@corp.nstu.ru

## Abstract

The paper presents an approach to organizing the requirements analysis based on the ontological model consisting of three OWL ontologies. The distinguishing features of the proposed approach are the combination of the ontological model with the rules formulated in the predicate form, as well as the focus on the support of the currently most demanded agile approach to software engineering. To take these aspects into account the proposed theoretical apparatus includes such important peculiarities as building and refining the Product backlog, tracing the requirements, identifying conflicting requirements and eliminating duplicate requirements. The latter functions are provided by applying logical inference to the system of axioms in the Prolog language obtained by combining the ontology of requirements, the ontology of the application domain and the formulated production rules. The rules are used for analysis of the relations between the requirements. In the current version, we determine four types of relations between a pair of the requirements that can be determined: isConflict, isDuplicate, isInterconnect, isNotInterconnect.

## 1 Introduction

Software development is a knowledge-intensive work in which team discussion of the current stage results in terms of their compliance with the requirements formulated in the requirements engineering stage is the key to success. In the process of requirements engineering inaccurate and incomplete ideas about a software product are transformed into a formal requirements specification . In agile environment, the backlog concept is used is part of an approach to working with the requirements. Backlog is a log of the remaining work that a team needs to perform. There is a product backlog and a sprint backlog [Dar2016]. Before planning a sprint, elements of the product backlog should be divided into small user stories, basic behavior scenarios should be written to them, implementation efforts should be assessed, and priorities should be determined. This work is done by the product owner together with the development team in preparation for the product backlog grooming. To work efficiently with requirements, this approach requires automated tools to support the requirements engineering process.

In: S. Hölldobler, A. Malikov (eds.): Proceedings of the YSIP-3 Workshop, Stavropol and Arkhyz, Russian Federation, 17-09-2019–20-09-2019, published at http://ceur-ws.org

Numerous studies have proven the effectiveness of using ontologies to support the requirements engineering process (see, for example, the surveys [Als2019] and [Der2014]). However, in various studies, ontological modeling is applied to different aspects of the multilateral and complicated process of requirements engineering. In the early 2000s, it was proposed to consider the process of developing ontologies as a subprocess of the requirements engineering process [Bre2003], whereas in [Zhu2005] the domain ontology is efficiently used as an infrastructure for specifying software requirements. An approach to automating the process of validation and measurement of knowledge about the requirements was proposed in [Sieg2014].

Later, the ontological approach began to be applied not only to the software engineering with a classic waterfall cycle, but also to the iterative development process. In [Tham2016] ontological approach is applied to improving the requirements engineering process in the Agile environment. The ontology is designed to work with user story templates. Ontology enables to identify synonymous concepts, hyperonymic and hyponymic relations between the concepts. The latter enables the requirements engineering process to define user stories that must be performed for user roles of applications that include other roles. In [Sitt2016] an approach to developing a recommender system that supports the evolution of the Agile requirements is presented. It is proposed to use the following four ontologies: "Environmental Context Ontology", "Problem Domain Ontology", "Requirements Ontology" and "Agile Requirements Ontology".

Issues of requirements traceability are addressed in the [Avd2015] devoted to the development of frame ontology which enables to create a consistent model of requirements types for a specific software development project.

In [Bha2015] two types of ontologies are distinguished which can be used to represent knowledge of the software domain being developed: the application domain ontology and the application domain feature model ontology. In [Mur2015] the requirements are considered as a specialized subset of a set of knowledge about the domain while the domain ontology is used as a "background source" in the process of extracting requirements for a software product from natural language texts.

The paper [Egy2004] addresses the issues of defining cooperation and conflicts between the requirements. Requirements conflict if they contain contradictory statements about common software attributes and are in a state of cooperation if they mutually supplement statements about attributes.

In [Rob2016] an approach to the automatic construction of an ontology from a set of user stories is proposed. For text processing in natural english language, the spaCy library is used, which allows for parsing sentences based on a dependency tree, searching for named groups.

Ontologies are used to formally represent the knowledge about the types of requirements, the structure of the requirements specification, as well as the subject area of the software product [Cas2010]. The trends of recent studies in the field of improving the requirements engineering process are the methods of working with requirements as with the facts from the application domain model. The relevance of the developed approach to extraction, automation and analysis of the requirements in natural language is determined by the variability of the requirements and the need for a quick comparison of the requirements texts.

Thus, summarizing the above, the following areas can be distinguished in the existing studies of the possibilities for using the ontologies in the field of requirements engineering: (1) development of the ontologies for the representation of knowledge about the requirements engineering process taking into account peculiarities of certain techniques, requirements types and characteristics of their quality, (2) development of ontologies for representing knowledge of the application domain which describe domain concepts, relationships between different users roles, actions of users in the system, components of the software product, (3) development of the requirements ontologies for identifying conflicts, duplicates and cooperation between the requirements. It is worth noting that if the first two areas are given quite a lot of attention in the literature, the third question is raised very superficially.

It is worth noting that while the first two areas are given quite a lot of attention in the literature, the third issue, which is very important from the point of view of obtaining really useful practical results, is touched on very superficially. This is explained by the complexity of the problem of formalizing knowledge and implementing the mechanism of knowledge inference. But the ontological approach gives maximum benefit only when the model described in terms of formal logic starts using the inference mechanism, which will make it possible in our case to identify conflicting requirements, to reveal the incompleteness or redundancy of the knowledge contained in the combined model of the requirements and the application domain.

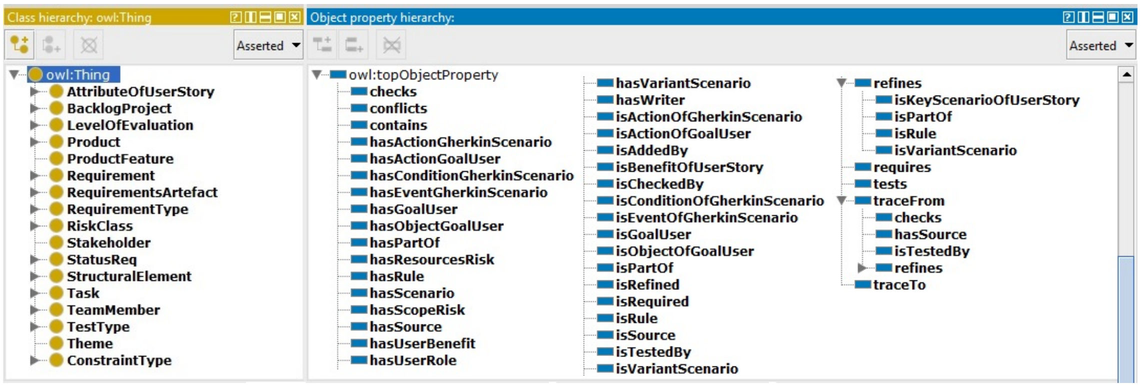
We propose an approach to agile requirements engineering based on combination of the OWL ontology with production rules followed by the system which allows you to apply full-scale inference to a system of logical axioms, uploaded to the Prolog inference system. The proposed ontology system includes three ontologies. The first ontology is necessary to represent knowledge of the engineering requirements process taking into account the peculiarities of the agile development. The second ontology is necessary to represent knowledge of the application domain. The third ontology is needed to identify the relations between requirements.

The rest of paper is organized as follows. Section 2 describes the OWL ontology system. Section 3 proposes an approach to converting textual requirements into the ontology. Section 4 describes the approach to analyzing the relationship between requirements. Section 6 summarizes the work.

## 2 OWL ontologies for the requirements engineering

### 2.1 Ontology for the requirements engineering in Agile

An ontology to support the requirements engineering process in Agile consists of classes that correspond to events, roles and agile artefacts. The ontology classes are instances of project requirements and their artefacts, as well as information about the development team and requirements sources . Figure 1 shows the taxonomy of the top-level classes and the object properties. A description of the top-level classes is given in Table 1.



```
Class hierarchy: owl:Thing                 Asserted
owl:Thing
    AttributeOfUserStory
    BacklogProject
    LevelOfEvaluation
    Product
    ProductFeature
    Requirement
    RequirementsArtefact
    RequirementType
    RiskClass
    Stakeholder
    StatusReq
    StructuralElement
    Task
    TeamMember
    TestType
    Theme
    ConstraintType

Object property hierarchy:                 Asserted
owl:topObjectProperty
    checks
    conflicts
    contains
    hasActionGherkinScenario
    hasActionGoalUser
    hasConditionGherkinScenario
    hasEventGherkinScenario
    hasGoalUser
    hasObjectGoalUser
    hasPartOf
    hasResourcesRisk
    hasRule
    hasScenario
    hasScopeRisk
    hasSource
    hasUserBenefit
    hasUserRole
    hasVariantScenario
    hasWriter
    isActionOfGherkinScenario
    isActionOfGoalUser
    isAddedBy
    isBenefitOfUserStory
    isCheckedBy
    isConditionOfGherkinScenario
    isEventOfGherkinScenario
    isGoalUser
    isObjectOfGoalUser
    isPartOf
    isRefined
    isRequired
    isRule
    isSource
    isTestedBy
    isVariantScenario
    refines
        isKeyScenarioOfUserStory
        isPartOf
        isRule
        isVariantScenario
    requires
    tests
    traceFrom
        checks
        hasSource
        isTestedBy
        refines
    traceTo
```

Figure 1: The taxonomy of the upper level classes and object properties

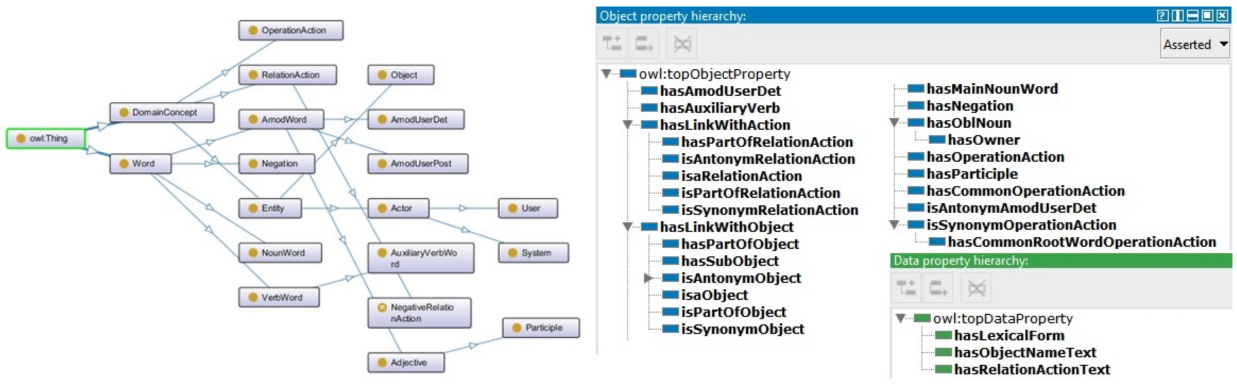
Table 1: The description of the upper-level classes

| Class name | Description |
|---|---|
| AttributeOfUserStory | describes the attributes of user stories such as priority, risk level, etc. |
| BacklogProject | includes subclasses describing the project's backlogs |
| LevelOfEvaluation | includes subclasses such as qualitative scales used in assessing priorities, risks, etc. |
| Product | contains subclasses corresponding to the concept "software product" |
| ProductFeature | instances of this class are the features being developed |
| Requirement | contains subclasses corresponding to the concept "requirement" |
| RequirementsArtefact | contains subclasses corresponding to the concept "requirements artefact" |
| RequirementType | includes subclasses describing the classification of requirement types |
| RiskClass | includes subclasses describing the classification of risks |
| Stakeholder | contains subclasses corresponding to the concept "stakeholder" |
| StatusReq | includes subclasses describing the classification of requirements statuses |
| StructuralElement | includes subclasses describing the structural elements of requirements |
| Task | includes instances which are tasks a development team works on |
| TeamMember | includes subclasses describing roles in a team |
| TestType | includes subclasses describing the classification of test types |
| ConstraintType | includes subclasses describing the types of constraints for non-functional requirements |

The object properties reflect the relations that can be established between instances of the ontology classes. For example, to specify the relation "the requirement refines another requirement", the object property "refines" is used. This object property, in turn, includes sub-properties that can be established between different classes of requirements artefacts which by their nature are also requirements (for example, the behavior scenario). It is proposed to use ontology for representing knowledge of traceability. Ontology includes two types of relations implemented through object properties: traceFrom and traceTo. The first relation includes subrelations which enables bottom-up tracing, the second top-down tracing. For the object property "traceFrom" and the sub-properties included in it, the inverse properties are set - "Inverse Of". The object property "conflicts" is specifed that requirements conflict with each other. This can be done directly in this ontology or transferred from the ontology for analyzing of relations between the requirements.

## 2.2 Application domain ontology

In this paper, it appears that the application domain ontology must include a set of facts about the subject area, a users role model, a features tree model. Terminology description of the application domain includes the domain concepts and words that make up the domain concepts. In this work, it is proposed to present a set of facts about the subject area taking into account some object properties and classes characteristic of lexical ontologies. This will allow later when analyzing data to compare data ontologies of the application domain ontology with data from external lexical ontologies. Words include nouns, verbs, auxiliary verbs and definitions combined with nouns. The hierarchical structure of the classes "DomainConcept" and "Word" is presented in Figure 2. Object and data properties are also presented in Figure2.The description of these classes is given in Table 2.



```
owl:topObjectProperty
    hasAmodUserDet
    hasAuxiliaryVerb
    hasLinkWithAction
        hasPartOfRelationAction
        isAntonymRelationAction
        isaRelationAction
        isPartOfRelationAction
        isSynonymRelationAction
    hasLinkWithObject
        hasPartOfObject
        hasSubObject
        isAntonymObject
        isaObject
        isPartOfObject
        isSynonymObject
    hasMainNounWord
    hasNegation
    hasOblNoun
        hasOwner
    hasOperationAction
    hasParticiple
    hasCommonOperationAction
    isAntonymAmodUserDet
    isSynonymOperationAction
        hasCommonRootWordOperationAction

owl:topDataProperty
    hasLexicalForm
    hasObjectNameText
    hasRelationActionText
```

Figure 2: A conceptual structure of "DomainConcept" and "Word", object and data properties

In the class "DomainConcept", there are two main subclasses that are needed to describe and verify the validity of requirements: "Entity" and "RelationAction". The class "Entity" in turn is divided into classes "Actor" and "Object". Actor is an entity with some behavior. An actor can be a user or a software system (subsystem). An object is an entity to which actions performed by an actor are directed. The actor and the object are interconnected by actions. Relations-actions, in turn, should include the semantic verb, and can also include the auxiliary verb and the not particle. For example: "can not delete." The verb "delete" in this phrase is semantic. Semantic verbs of the subject area will be allocated in a separate class "OperationAction".

The class "User" is a subclass of the class "Actor". To build a role model, it is necessary to distinguish user classes that are typical for the subject area, the object properties between class instances, and their data properties. Let us consider the approach to building in ontology of the users role model for the company's web-site designed to work with company consumers of the products or services (clients). The following classes of users can be distinguished here: internal users of the system (employees) and external users (clients). All internal users of the system must be registered users, and consumers of products or services produced by an enterprise may be registered users who work with the web system through a personal account. External users can be registered and unregistered. On the other hand, the user of the web system can be: (1) a registered user who works through a personal account (logged in user) and (2) a user unregistered in the system who works with the publicly accessible part of the web system.

A user registered in the system but not logged into the personal account at a given time and who works with the public part of the web system (an unauthorized user).

In turn, an unregistered user can be a user who has an active account in the system (a user who has logged off in the system) and a user who does not have an account. For any non-logged-on user of the web system, features is available corresponding to the publicly accessible part of the system intended for the client.

Table 2: Classes for describing domain concepts

| Class name | Description |
|---|---|
| DomainConcept | class whose elements are domain concepts |
| Entity | class whose elements are entities |
| Actor | class whose elements are actors |
| System | class whose elements are actors-systems |
| User | class whose members are actors-users |
| Object | class whose elements are objects used by actors |
| OperationAction | class whose elements are semantic verbs |
| RelationAction | class whose elements are actions |
| NegativeRelationAction | class whose elements are negative actions |
| Word | class whose elements are words |
| AmodWord | class whose elements are words defining nouns |
| Adjective | class whose elements are adjectives |
| Participle | class whose elements are participles |
| AmodUserDet | class whose elements are words that define the ownership of an object to a user |
| AmodUserPost | class whose elements are words defining the level of the user's position |
| Negation | class containing the particle "NOT" |
| NounWord | class whose elements are nouns |
| VerbWord | class whose elements are verbs |
| AuxiliaryVerbWord | class whose elements are auxiliary verbs |

Based on this, a class structure was designed to describe the users role model. The hierarchical structure of the class "User" is presented in Figure 3. Object and data properties are also presented in Figure 3.

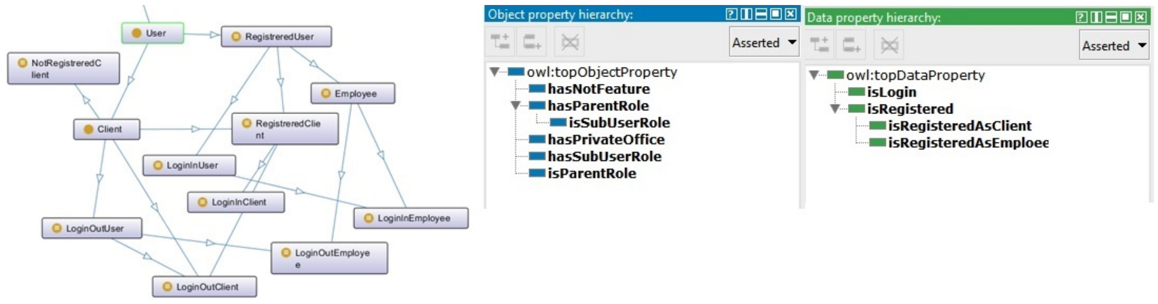


Figure 3: A conceptual structure of "User", object and data properties

The description of data and object properties for the class "User" is given respectively in Tables 3 and 4.

The class "ProductFeatures" is used to build a feature tree. Feature can be mandatory or optional. The description of feature variability is incorrect if optional feature has mandatory child features. On the Figure 4 shows the axioms for evaluation of feature variability. Object and data properties of class "ProductFeatures" are also presented in Figure 4 .

## 2.3 Ontology for analysis of relations between the requirements

To determine the relations between sentences with requirements, it is necessary to transform them into ontology instances and establish relations between individual elements of sentences, such as an actor, an action, and an object. Figure 5 shows the object properties between instances of ontology classes, their domains and ranges.

Object properties between instances of the same class "Actor" are grouped into a object property class "relationActor". They are also grouped into object property classes "relationAction", "relationObject" and "relationrequirement" relations between the classe instances "Action", "Object" and "Requirement", respectively.

Object properties "sameAsActor", "sameAsAction", "sameAsObject" are set between instances of the corresponding classes "Actor", "Action" and "Object" if one name is used for elements of two requirements or, for

Table 3: Description of data properties of individuals of class "User"

| Relation name | Relation Properties | Range | Subproperty |
|---|---|---|---|
| isLogin | Functional | xsd:boolean | topDataProperty |
| isRegistered name | - | xsd:boolean | topDataProperty |
| isRegisteredAsClient | Functional | xsd:boolean | isRegistered |
| isRegisteredAsEmploee | Functional | xsd:boolean | isRegistered |

Table 4: Description of object properties of individuals of class "User"

| Relation name | Relation Properties | Inverse Of | Subproperty |
|---|---|---|---|
| hasParentRole | Transitive | isParentRole | topObjectProperty |
| hasSubUserRole | Asymmetric, Irreflexive | isSubUserRole | isParentRole |
| isParentRole | Transitive | hasParentRole | topObjectProperty |
| isSubUserRole | Asymmetric, Irreflexive | hasSubUserRole | hasSubUserRole |
| hasNotFeature | Asymmetric, Irreflexive | - | topObjectProperty |

example, the full name in one and the abbreviation in the second. These object properties are symmetric. Also, object properties "sameAsActor", "sameAsAction" and "sameAsObject" are established between instances of classes if the connection of "Same Individual As" with another individual is specified in the application domain ontology in the description of the class instance . For example, in the role model of system user between instances "guest" and "visitor" the link "Same Individual As" is set.

Object properties "isaActor", "isaAction" and "isaObject" are used to establish hierarchical relation. For example, the "senior manager" is the "manager".

Object properties "synonymsActor", "synonymsAction" and "synonymsObject" are set if the values of the corresponding requirements elements have a synonymous value. For example, the action "to view" records is synonymous with the action "to list". These object properties can be get from the application domain ontology or from external sources, for example, the open electronic thesaurus of the russian language YARN.

Object properties "antonymsAction" and "antonymsObject" are set between instances of the corresponding classes if they have the opposite meaning. For example, the object "your comment" and the object "someone else's comment".

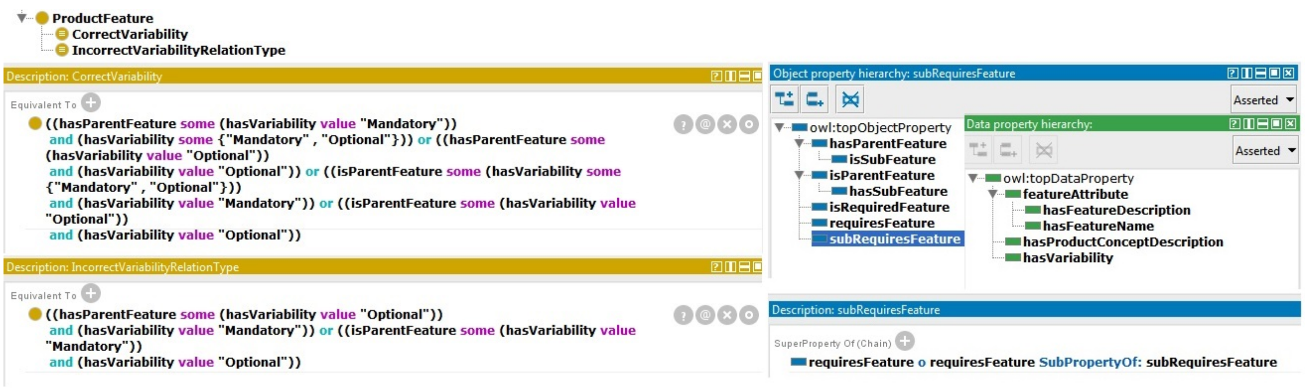
Object properties "partOfAction" and "partOfObject" are set up between instances of the corresponding classes if one is part of the other. For example, the action "delete" is part of the action "edit".

In all other cases, the corresponding requirements elements are considered to be related by the object properties "no-relationActor", "no-relationAction" and "no-relationObject". In other words, the relations between them is not revealed.

## 3 Converting textual requirements into OWL ontology

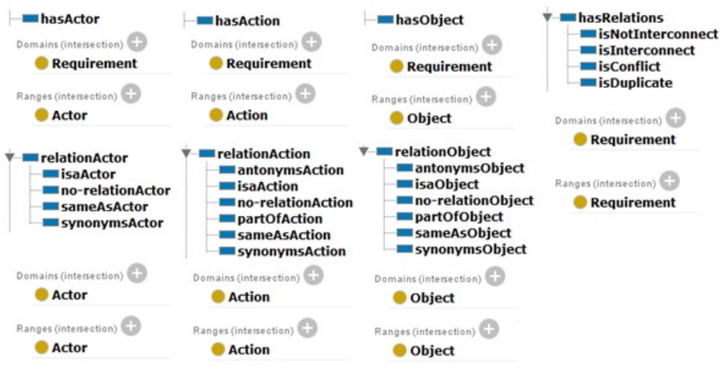
The conversion of textual requirements into ontology instances can be carried out in manual and semi-automatic mode using automatic text processing tools. The UDPipe parser with the Russian language model can be used for text processing of requirements in Russian. This tools parses the sentence and gives the results in the format CoNLL-U. The following fields are used to describe the annotation of text token (word, punctuation mark, or special character) of a sentence:

1) ID: token index
2) FORM: word form or punctuation symbol
3) LEMMA: lemma of word form
4) UPOSTAG: universal part of speech tag
5) XPOSTAG: language tag of a part of speech (if not, a dash is indicated)
6) FEATS: list of morphological features from the universal feature inventory or from a specific language extension



```
ProductFeature
    CorrectVariability
    IncorrectVariabilityRelationType

Description: CorrectVariability
Equivalent To
  ((hasParentFeature some (hasVariability value "Mandatory"))
   and (hasVariability some {"Mandatory" , "Optional"})) or ((hasParentFeature some
  (hasVariability value "Optional"))
   and (hasVariability value "Optional")) or ((isParentFeature some (hasVariability some
  {"Mandatory" , "Optional"}))
   and (hasVariability value "Mandatory")) or ((isParentFeature some (hasVariability value
  "Optional"))
   and (hasVariability value "Optional"))

Description: IncorrectVariabilityRelationType
Equivalent To
  ((hasParentFeature some (hasVariability value "Optional"))
   and (hasVariability value "Mandatory")) or ((isParentFeature some (hasVariability value
  "Mandatory"))
   and (hasVariability value "Optional"))

Object property hierarchy: subRequiresFeature                Asserted
owl:topObjectProperty
    hasParentFeature
        isSubFeature
    isParentFeature
        hasSubFeature
    isRequiredFeature
    requiresFeature
    subRequiresFeature

Data property hierarchy:                                     Asserted
owl:topDataProperty
    featureAttribute
        hasFeatureDescription
        hasFeatureName
    hasProductConceptDescription
    hasVariability

Description: subRequiresFeature
SuperProperty Of (Chain)
  requiresFeature o requiresFeature SubPropertyOf: subRequiresFeature
```

Figure 4: Object and data property for "ProductFeature", axioms for evaluation of feature variability



```
hasActor                  hasAction                 hasObject                 hasRelations
Domains (intersection)    Domains (intersection)    Domains (intersection)        isNotInterconnect
  Requirement               Requirement               Requirement                 isInterconnect
Ranges (intersection)     Ranges (intersection)     Ranges (intersection)         isConflict
  Actor                     Action                    Object                      isDuplicate
                                                                              Domains (intersection)
                                                                                Requirement
relationActor             relationAction            relationObject            Ranges (intersection)
    isActor                   antonymsAction            antonymsObject          Requirement
    no-relationActor          isaAction                 isaObject
    sameAsActor               no-relationAction         no-relationObject
    synonymsActor             partOfAction              partOfObject
                              sameAsAction              sameAsObject
                              synonymsAction            synonymsObject
Domains (intersection)    Domains (intersection)    Domains (intersection)
  Actor                     Action                    Object
Ranges (intersection)     Ranges (intersection)     Ranges (intersection)
  Actor                     Action                    Object
```

Figure 5: Object properties of ontology, their domains and ranges

7) HEAD: number of the parent token or zero for the root token
8) DEPREL: universal dependency relation to parent token (set to "root" if HEAD = 0)
9) DEPS: list of secondary dependencies
10) MISC: additional information

Actor, action and object can be automatically extracted from the results of the analysis of the proposal. At the same time, the sentence with the requirement should be formulated simply enough. In other words, sentence must not contain a lot of speech turns. Figure 6 illustrates the dependency tree of the sentence: "As a manager, I can only delete the product descriptions I added".

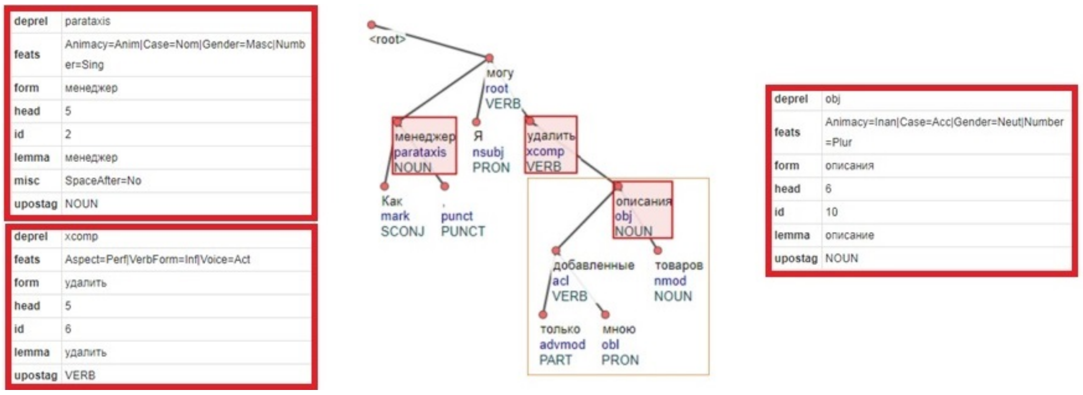
Production rules can be applied to process the results of parsing sentences. To extract "Entity" , it is necessary to analyze tokens that are defined as nouns (UPOSTAG = NOUN). For example, the main word in the name of an entity can be defined by the rule:

**IF** $X$ is a noun **AND** $X$ is related by an dependency relation $R_{XZ}$ with the verb $Z$
**THEN** $X$ is the main word in the name of the entity.

Next, it is determined whether the given noun has a definition in order to extract the name of the entity as a whole. Further, to determine the composite names of entities, the rules for analyzing dependencies are applied, taking into account dependencies between tokens, their parts of speech and their morphological characteristics in the russian model *Russian-syntagrus-ud-2.4-190531.udpipe*. Entities, in turn, are divided into actor and objects. The main word in the name of an entity that is a user is an animated noun (UPOSTAG = NOUN and FEATS: Animacy = Anim).

To extract the "action", it is necessary to analyze the tokens that are defined as verbs (UPOSTAG = VERB). To define an operation-action (a semantic verb for a relationship-action), the following rule applies:

**IF** $Y$ is a verb **AND** $Y$ is not an auxiliary verb **THEN** $Y$ is an operation-action.



| | |
|---|---|
| deprel | parataxis |
| feats | Animacy=Anim\|Case=Nom\|Gender=Masc\|Number=Sing |
| form | менеджер |
| head | 5 |
| id | 2 |
| lemma | менеджер |
| misc | SpaceAfter=No |
| upostag | NOUN |

| | |
|---|---|
| deprel | xcomp |
| feats | Aspect=Perf\|VerbForm=Inf\|Voice=Act |
| form | удалить |
| head | 5 |
| id | 6 |
| lemma | удалить |
| upostag | VERB |

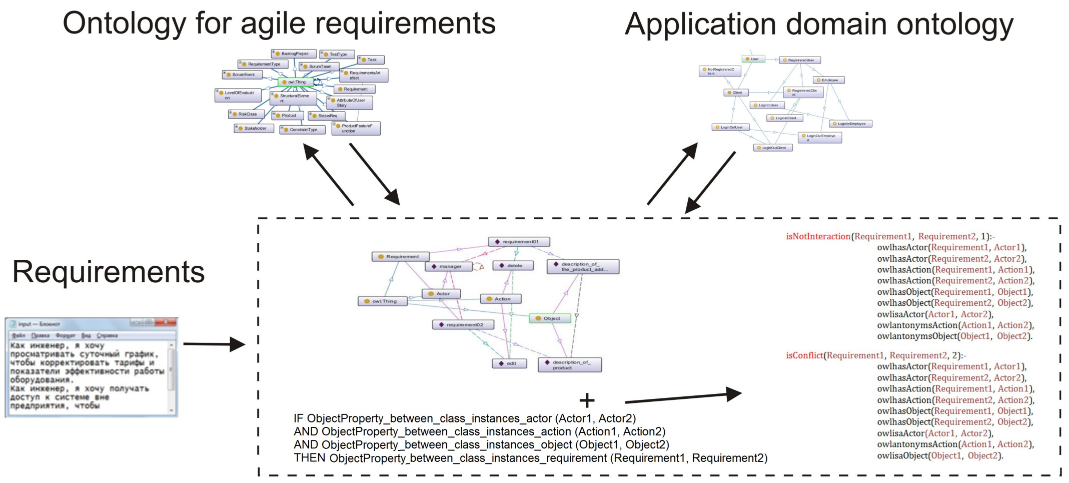
| | |
|---|---|
| deprel | obj |
| feats | Animacy=Inan\|Case=Acc\|Gender=Neut\|Number=Plur |
| form | описания |
| head | 6 |
| id | 10 |
| lemma | описание |
| upostag | NOUN |

Figure 6: An example of dependency tree obtained by UDPipe

Next, it is necessary to determine if this operation-action has a particle "NOT" and form an instance of class "Action".

## 4 An approach to analyzing of relations between the requirements

Requirements are converted from textual requirements to an ontology for analyzing of relations between the requirements. If necessary, data on previously verified requirements from the main ontology of the project are transferred to the same ontology. Further, relations between instances of classes "Actor", "Action" and "Object" are established. Production rules are then applied to determine the relations between requirements. All statements are converted to rules and facts in the SWI-Prolog language. Software modules for processing OWL files, building a production model and transferring facts and rules to the Prolog system are implemented in Python. The general scheme for organizing requirements analysis is shown in Figure 7.



Как инженер, я хочу просматривать суточный график, чтобы корректировать тарифы и показатели эффективности работы оборудования.
Как инженер, я хочу получать доступ к системе вне предприятия, чтобы

```
IF ObjectProperty_between_class_instances_actor (Actor1, Actor2)
AND ObjectProperty_between_class_instances_action (Action1, Action2)
AND ObjectProperty_between_class_instances_object (Object1, Object2)
THEN ObjectProperty_between_class_instances_requirement (Requirement1, Requirement2)
```

```
isNotInteraction(Requirement1, Requirement2, 1):-
                    owlhasActor(Requirement1, Actor1),
                    owlhasActor(Requirement2, Actor2),
                    owlhasAction(Requirement1, Action1),
                    owlhasAction(Requirement2, Action2),
                    owlhasObject(Requirement1, Object1),
                    owlhasObject(Requirement2, Object2),
                    owlisaActor(Actor1, Actor2),
                    owlantonymsAction(Action1, Action2),
                    owlantonymsObject(Object1, Object2).

isConflict(Requirement1, Requirement2, 2):-
                    owlhasActor(Requirement1, Actor1),
                    owlhasActor(Requirement2, Actor2),
                    owlhasAction(Requirement1, Action1),
                    owlhasAction(Requirement2, Action2),
                    owlhasObject(Requirement1, Object1),
                    owlhasObject(Requirement2, Object2),
                    owlisaActor(Actor1, Actor2),
                    owlantonymsAction(Action1, Action2),
                    owlisaObject(Object1, Object2).
```

Figure 7: A scheme of the analyzing of relations between the requirements

The following scheme for generating production rules of the form is used to determine the type of relation between requirements:

**IF** ObjectProperty_between_class_instances_actor (Actor1, Actor2)
**AND** ObjectProperty_between_class_instances_action (Action1, Action2)
**AND** ObjectProperty_between_class_instances_object (Object1, Object2)
**THEN** ObjectProperty_between_class_instances_requirement (Requirement1, Requirement2)

Depending on the used rule one of the following four relations between the requirements are set: isConflict, isDuplicate, isInterconnect, isNotInterconnect.

Consider an example of comparing two requirements on the following sentences:

*User story 01:* Как менеджер, Я могу удалить только добавленные мною описания товаров

As a manager, I can only delete the product descriptions I added

*User story 02:* Как менеджер, Я могу редактировать описания всех товаров из каталога

As a manager, I can edit descriptions for all products from the catalog

Figure 8 shows the relationships between instances of classes "Object" and "Actions" of the application domain ontology.

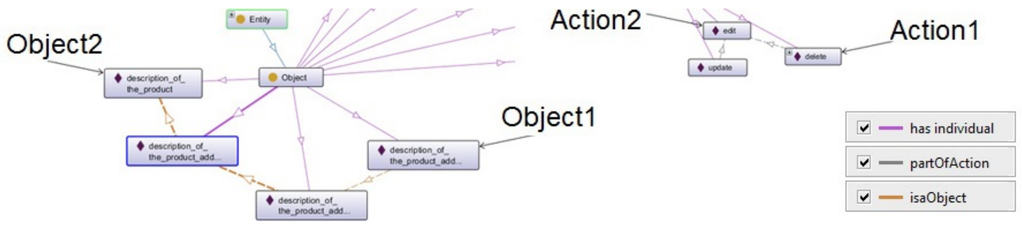
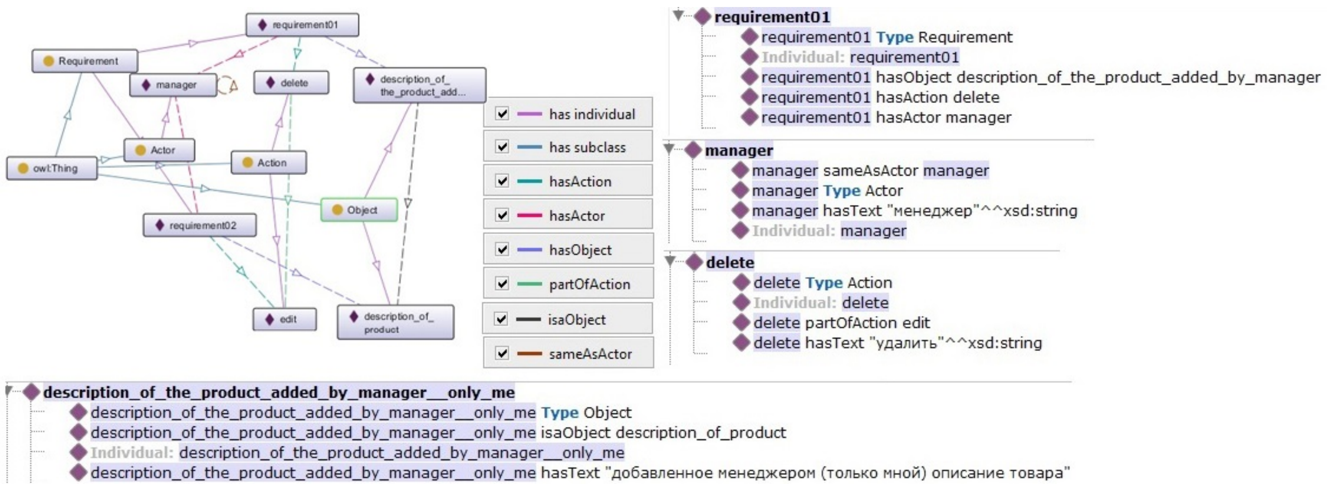

Object2
Entity
Object
description_of_the_product
description_of_the_product_add...
description_of_the_product_add...
description_of_the_product_add...
Object1
Action2
edit
update
delete
Action1
has individual
partOfAction
isaObject

Figure 8: Fragment of application domain ontology

Figure 9 shows the transformation of text data into ontology instances for analyzing requirements. Relations between instances of the classes "Actor", "Action" and "Object" are established based on application domain ontology.



**requirement01**
- requirement01 Type Requirement
- Individual: requirement01
- requirement01 hasObject description_of_the_product_added_by_manager
- requirement01 hasAction delete
- requirement01 hasActor manager

**manager**
- manager sameAsActor manager
- manager Type Actor
- manager hasText "менеджер"^^xsd:string
- Individual: manager

**delete**
- delete Type Action
- Individual: delete
- delete partOfAction edit
- delete hasText "удалить"^^xsd:string

**description_of_the_product_added_by_manager__only_me**
- description_of_the_product_added_by_manager__only_me Type Object
- description_of_the_product_added_by_manager__only_me isaObject description_of_product
- Individual: description_of_the_product_added_by_manager__only_me
- description_of_the_product_added_by_manager__only_me hasText "добавленное менеджером (только мной) описание товара"

Figure 9: Fragment of requirements ontology

The following result will be obtained after data is inserted into the rule:

**IF** sameAsActor("менеджер" , "менеджер") **AND** partOfAction ("удалить" , "редактировать")
**AND** isaObject("добавленное менеджером (только мною) описание товара" , "описание товара в каталоге")
**THEN** isConflict (Requirement1, Requirement2).

These requirements conflict as the product descriptions added by a separated manager are a subset of all product descriptions. The operation "edit" includes the operation "delete". Therefore, according to the first statement, the manager can delete any elements of the set "product description" and, according to the second, only some of them.

## 5 Conclusion

The paper proposed an approach to analyzing requirements based on ontological and production models of knowledge representation. The ontology structure was designed for storing knowledge of the progress of work with requirements with an agile approach. Designed ontologies are designed to solve typical problems for requirements engineering in agile software development. These include refining the Product backlog, tracing requirements, identifying conflicting requirements, eliminating duplicate requirements. It was proposed to extract from the textual requirements of the actor the action and the object, and to save them in the form of appropriate ontology instances. An approach to the formation of production rules for the analysis of relations between instances of ontology classes is defined. For the successful automatic extraction of instances of ontology classes "Actor", "Action" and "Object" from textual requirements, it is required to record the requirements in the form of simple sentences.

## 6 Acknowledgements

The reported study was funded by Russian Ministry of Education and Science, according to the research project No. 2.2327.2017/4.6.

## References

[Dar2016] N. D. Darwish, S. Megahed. *Requirements Engineering in Scrum Framework – Volume 149 / International Journal of Computer Applications* . 2016. P.24-29.

[Als2019] A. A. Alsanad, A. Chikh, A. Mirza. *A Domain Ontology for Software Requirements Change Management in Global Software Development Environment – Volume 7 / IEEE Access* . 2019. P.49352-49361.

[Der2014] D. Dermeval, J. Vilela, I. I. Bittencourt, J. Castro, S. Isotani, P. Brito. *A Systematic Review on the Use of Ontologies in Requirements Engineering / Brazilian Symposium on Software Engineering* . 2014. P.1-10.

[Bre2003] K. K. Breitman, J. C. S. Prado Leite. *Ontology as a Requirements Engineering Product / International Requirements Engineering Conference.* 2003. P.309-319.

[Zhu2005] X. Zhu. *TInconsistency Measurement of Software Requirements Specifications: An Ontology-Based Approach / Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems.* Washington, 2005. P.402-410.

[Sieg2014] K. Siegemund. *Contributions To Ontology-Driven Requirements Engineering : dissertation to obtain the academic degree Doctoral engineer (Dr.-Ing.)* . Dresden: Technischen Universitat Dresden, 2014. 236 p.

[Tham2016] C. Thamrongchote , W. Vatanawood . *Business process ontology for defining user story / 15th International Conference on Computer and Information Science (ICIS).* 2016.

[Sitt2016] S. Sitthithanasakul, N. Choosri. *Using ontology to enhance requirement engineering in agile software process / 10th International Conference on Software, Knowledge, Information Management and Applications (SKIMA).* 2016. P.181-186.

[Avd2015] T. V. Avdeenko, N. V. Pustovalova. *The ontology-based approach to support the completeness and consistency of the requirements specification – Volume 9 / International Siberian conference on control and communications (SIBCON2015).* Omsk: IEEE, 2015.

[Bha2015] M. P. S. Bhatia, A. Kumar, R. Beniwal. *Ontologies for Software Engineering: Past, Present and Future – Volume 9 / Indian Journal of Science and Technology.* 2015.

[Mur2015] S. Murugesh, A. Jaya. *Construction of Ontology for Software Requirements Elicitation – Volume 8 / Indian Journal of Science and Technology.* 2015.

[Egy2004] A. Egyed, P. Grunbacher. *Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help – Volume 21 / IEEE Software.* 2004. P.50-58.

[Rob2016] M. Robeer, G. Lucassen, J. M. E. M. van Der Werf, F. Dalpiaz, S. Brinkkemper . *Automated Extraction of Conceptual Models from User Stories via NLP / 24th International Requirements Engineering (RE) Conference.* 2016. P.196-205.

[Cas2010] V. Castaneda , L. Ballejos , M. L. Caliusco , M. R. Galli. *The use of ontologies in requirements engineering – Volume 10 / Global Journal of Research In Engineering* . 2010. P.2-8.