Seventh Framework Programme

CallFP7-ICT-2013-10

Project Acronym:	**S-CASE**
Grant Agreement N^{o}:	**610717**
Project Type:	**COLLABORATIVE PROJECT**
Project Full Title:	**Scaffolding Scalable Software Services**

D3.1.2 Module for extracting software artefacts from text

Nature:	**R**
Dissemination Level:	**PU**
Version #:	**1.0**
Date:	**30 July 2014**
WP number and Title:	**WP3 Multimodal Information Processing**
Deliverable Leader:	**UEDIN**
Author(s):	**Michael Roth (UEDIN), Ewan Klein (UEDIN), Themistoklis Diamantopoulos (AUTH)**
Revision:	**Davide Tossi (INS), Luigi Lavazza (INS), Andreas Symeonidis (AUTH)**
Status:	**Submitted** (Draft, Peer-Reviewed, Submitted, Approved)

Document History

Version[1]	Issue Date	Status[2]	Content and changes
0.1	20 May 2014	Draft	TOC
0.2	29 May 2014	Draft	Revised TOC
0.3	7 July 2014	Draft	First draft sections 1, 3
0.4	8 July 2014	Draft	First draft sections 2, 4
0.5	9 July 2014	Draft	Completed first draft
0.7	21 July 2014	Draft	Revised draft sections 1, 3 based on review
0.8	22 July 2014	Draft	Revised draft sections 2, 4 (except 4.1) based on review
0.9	22 July 2014	Draft	Completed revised draft
1.0	30 July 2014	Final	Incorporated all reviewers' comments and submitted deliverable

Peer Review History[3]

Version	Peer Review Date	Reviewed By
0.6	14 July 2014	Davide Tosi (INS), Luigi Lavazza (INS)
1.0	30 July 2014	Andreas Symeonidis (AUTH)

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610717.

[1] Please use a new number for each new version of the deliverable. Use "0.#" for Draft and Peer-Reviewed. "x.#" for Submitted and Approved", where x>=1.Add the date when this version was issued and list the items that have been added or changed.

[2] A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.

[3]Only for deliverables that have to be peer-reviewed

Table of contents

Abbreviations and Acronyms

FR	Functional Requirement
OWL	Web Ontology Language
SVO	Subject-Verb-Object
TTL	Terse RDF Triple Language
SRL	Semantic Role Labelling
RDF	Resource Description Framework
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

Executive Summary

The module for extracting software artefacts from text converts informal requirements expressed in natural language into a formal language that maps to the S-*CASE* ontology. This is achieved by using natural language processing techniques to automatically parse and "understand" the unstructured textual input.

The semantic parser is developed on the basis of supervised optimization methods from machine learning and hence relies on annotated data as training material. The concept ontology is integrated as background knowledge and provides guidance for automatically detecting and classifying instances of concepts and relations in previously unseen text.

In the context of the S-CASE architecture, this module provides initial analyses of software requirements, which will be used in conjunction with information from UML diagrams and storyboards to populate the S-CASE registry, and will serve as the basis for a query mechanism that goes beyond keyword search.

This deliverable describes the various components of the module developed for the task outlined above, including (1) a concept ontology that defines a hierarchy of concepts and relations for representing software requirements, (2) a semantic parser that automatically maps text fragments to instances of concepts and relations defined in the ontology, (3) annotation guidelines and a web-based annotation tool and (4) a data set of annotated requirements used for training the parser.

1 Introduction

Deliverable 3.1 (Module for extracting software artefacts from text) describes the software components implemented for semi-automatically converting unstructured requirements written in natural language to a formal representation that maps to the S-CASE ontology. This deliverable is part of work package 3 (WP3), which aims to extract requirements from multi-modal input.

1.1 WP3 Objectives

The main goal of WP3 is to design the mechanisms for efficiently extracting requirements from formal models such as UML diagrams, as well as from text and images. Additionally, WP3 will design and implement the Question-Answering mechanism that will serve as the user interface for querying on software artefacts. The WP has four specific objectives:

- To recognize software requirements informally expressed in unstructured and semi-structured English text and provide them with formal semantics (T3.1).
- To analyse storyboards of intended user interactions with software (T3.2).
- To transform XMI-based UML diagrams into the S-CASE ontology and to semantically analyse images of UML diagrams (T3.3).
- To develop a question answering system that will allow developers to pose queries in natural language about the software components in the S-CASE repository (T3.4).

This deliverable focuses on the first objective. We describe the scope of the corresponding task in more detail in the following sub-section.

1.2 Scope of Task 3.1

This deliverable reports on work performed for Task 3.1, which comprises of the following sub-tasks: analysis of existing corpora of natural language software requirements, construction of a parser prototype that converts informal requirements into a formal language, and creation of a semantically annotated evaluation data set. Work on these tasks has resulted in the following contributions described in this deliverable: (1) a concept ontology that defines concepts and relations that describe static functionalities of a software system, (2) a semantic parser that extracts instances of such concepts and relations from text, and (3) an annotation tool that can be utilized by S-CASE users to annotate and revise semantic information expressed in and extracted from natural language text.

1.3 Structure of this Deliverable

The document is structured as follows. Section 2 describes the concept ontology developed for formally representing semantic information expressed in textual descriptions of software requirements. Section 3 provides information on the semantic parser that we developed for performing the extraction task automatically. Section 4 describes an initial data set of software requirements and an annotation tool developed to manually annotate and revise mappings between text fragments and semantic information. We conclude this document in Section 5 with a summary of our results to date.

2 Ontology for Software Requirements

This Section concerns the design of an ontology for storing information derived from functional requirements. This ontology shall provide a representation of the static view of the system, including functional requirements, use case diagrams and generally any static information derived from other types of input (e.g. analysis class diagrams). The first subsection provides some essential background knowledge on ontology languages, and the following ones present the ontology and illustrate its instantiation using examples.

2.1 Background on Ontology Languages and Notation

Ontologies provide a structured means of storing information. They are known to be particularly useful for storing linked data (i.e. data connected with relations) and they provide effective ways of retrieving stored data via queries. Although ontologies are traced as early as in the 1990s, their widespread usage was connected to the emergence of the World Wide Web.

Information in the World Wide Web can be represented in a variety of languages; a general-purpose language for representing such information is provided by the *Resource Description Framework (RDF)*. Mostly oriented towards representing metadata, the basic RDF data model has three main object types: resources, properties, and statements. A resource is any "thing" that is described by the language, while properties are relations among resources. Resources are unique, identified by a Uniform Resource Identifier (URI), while their values can be either simple string values or other resources. Finally, the resources and the properties are assigned values using statements.

Although the RDF data model provides a powerful conceptual framework, it defines no syntax for the language itself. Thus, RDF models are usually used along with the well-known *eXtensible Markup Language (XML)*. Additionally, both the RDF model and the XML require their respective schemas. The RDF schema defines the required hierarchies for the resources and the properties of the data model, e.g. a resource of type `dog` would be a *subresource* of resource `animal`. The XML schema is simply a way to restrict the structure of XML documents.

Although the combination of RDF and XML are powerful enough for storing and presenting information, processing the stored information is hard. To overcome this difficulty, the *Web Ontology Language (OWL)* has been designed in order to enhance the aforementioned model by incorporating semantic information and providing additional formal vocabulary. In accordance with RDF, OWL has *classes*, *properties*, and *statements*, while also including more advanced features such as cardinality or symmetry between properties.

In the context of S-CASE, we decided to use OWL since it is a well-known established standard of current research and industry communities. In addition, we use Protégé for visualizing and designing our ontology (Protégé, 2014) since it is also a well-known tool. For an extensive review of OWL languages and tools, the reader is referred to the deliverable 4.1 of WP4.

The visualizations used throughout this deliverable include OWL classes, properties, and individuals. Classes and individuals are drawn as rounded squares (with different colors), and properties are drawn as arrows. The shapes and arrows have labels that hold the name of each class or property. Note, however, that the has_subclass property, which defines the hierarchical nature of OWL classes is given unlabelled in order to avoid cluttering the visualizations. Instead, the arrow of has_subclass is continuous so that it is clearly distinguishable with respect to the other property arrows that are dashed.

2.2 Ontology Overview

Since the ontology must cover the static functional aspects of the system, its design was mainly focused on the simple concept of an acting unit of the system (e.g. actor or system) performing some action(s) on some object(s). This representation not only covers the main functionality of the system but it is also suitable for representing functional requirements. In specific, well-formed subject-verb-object (SVO) sentences are easy to model. In addition, although the main focus of this deliverable is functional requirements, use cases can also be modeled effectively, since they also consist of an actor-acts-on-object structure.

2.2.1 Ontology Class Hierarchy

The class hierarchy of the ontology is shown in Figure 1. As shown in that Figure, anything entered in the ontology (any owl:Thing) is actually a Concept. Instances of class Concept are further divided in the types of Project, Requirement, ThingType, and OperationType.

Project refers to the project analyzed while Requirement stores each functional requirement of the system. These two types are useful for instantiating the ontology while keeping the structure reversible. Since each project has several requirements and each requirement has several other concepts (see next subsection for relations), one can reconstruct the main structure of each project including each one of the requirements with the respective concepts.

ThingType and OperationType are the main types of objects found in any functional requirement. The former refers to acting units and units acted upon, while the latter involves all types of actions performed by the acting units on other objects. In specific, a ThingType instance can be one of the following classes:

- actor: refers to the actors of the project. It includes three types of subclasses:
  - useractor: the users of the system
  - external_system: any external systems interacting with the systemc
  - system: the system itself is also an actor
- object: involves any object or resource of the system that receives some action. Since the nature of some objects can be composite (or generally have some notion of transitiveness), the concept of receiving an object from a composite object or sending some object to a composite object has to be modeled. Thus, three subclasses of object are defined:
  - theme: the main subclass of object, involving any generic object or resource of the system
  - source: involves objects that are sources of other objects. For example, for the phrase “get tag from bookmark”, “tag” would be mapped as theme and bookmark as source.

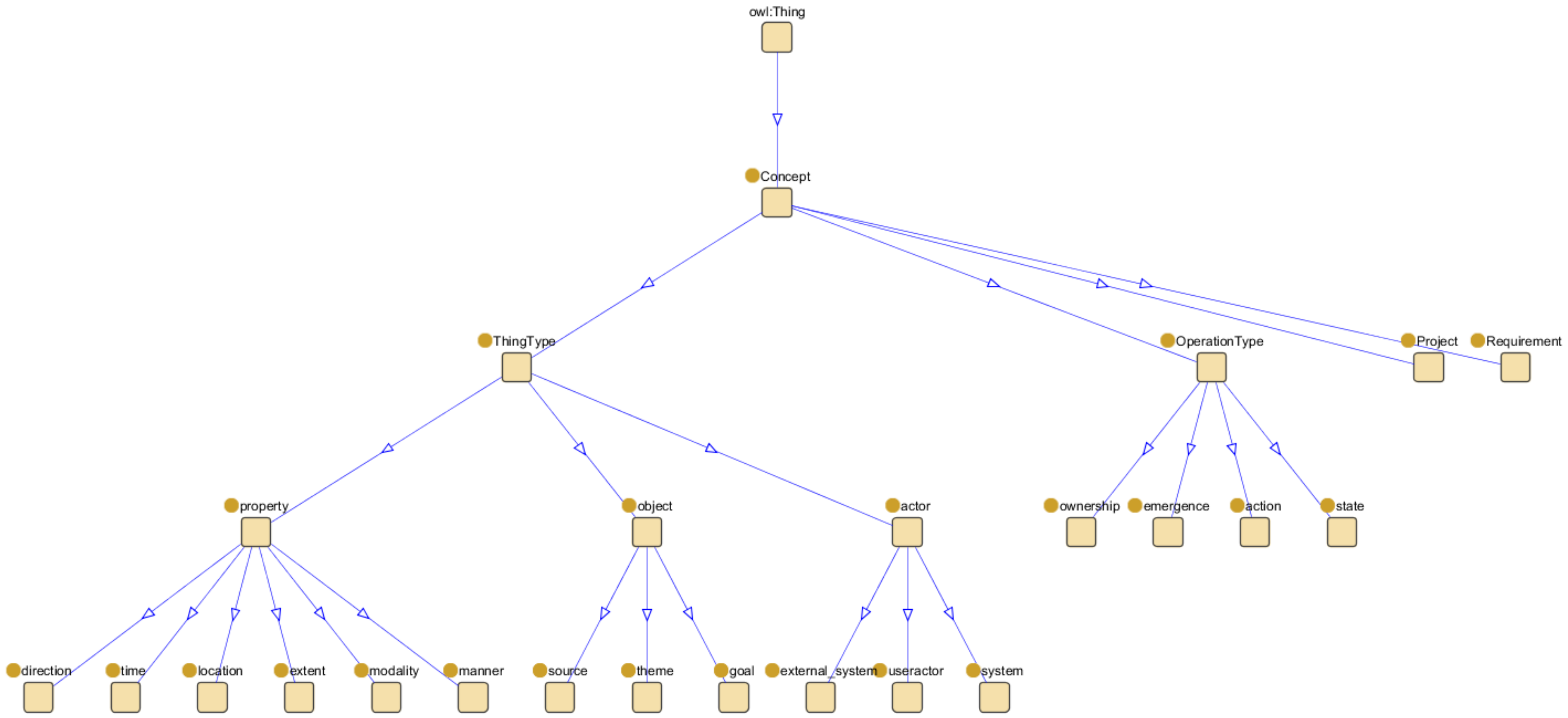


Figure 1 Ontology Class Hierarchy

    - `goal`: has the opposite meaning of `source`. It involves mapping objects that are destinations of other objects (e.g. via composition). For example, for the phrase "assign tag to bookmark", "tag" would be mapped as `theme` and bookmark as `goal`.

- `property`: includes all modifiers of objects or actions that assign some property to an object or to the action involved. Since instances of this class are highly generic (modifiers could concern many different properties of an object), several subclasses are defined to disambiguate among the modifier types with different semantics. The main intuition behind these properties came from the PropBank project (Palmer et al., 2005). The subclasses are shown below:
    - `direction`: includes modifiers that specify a notion along some path. For example, in the requirement "The user must be able to navigate North", "North" is mapped as a `direction`.
    - `time`: includes modifiers that show when actions take place. In most cases, temporal modifiers provide conditional constructs and/or successive views of requirements. For example, in the requirement "The system must be able to return to the menu when a movie is playing", the phrase "when a movie is playing" is a `property` of type `time`.
    - `location`: includes modifiers that indicate where some action takes place. For example, in the requirement "On the main page, the user must be able to exit", the phrase "on the main page" is a `location`.
    - `extent`: includes modifiers that indicate an amount of change. Changes may be expressed in terms of numbers, quantifiers or comparatives. For example, in the requirement "The system must automatically exit if no action is performed for 30 minutes", the phrase "for 30 minutes" is a `property` of type `extent`. Although `extent` is probably more typical for non-functional requirements, functional requirements may also contain such properties.
    - `modality`: includes constructs containing auxiliary verbs. Indicative examples include e.g. "a bookmark that can be deleted", or "a query that will not be stored".
    - `manner`: includes constructs (mainly adverbs) that specify how an action is performed. For example, in the requirement "The user must be able to search a POI by name", the "by name" is of type `manner`.

Finally, the class `OperationType` includes all operations performed by a user, either transitive or not. Thus, the subclasses of `OperationType` are:

- `ownership`: involves operations that express possession. In functional requirements, these operations are commonly expressed using the verb "have". For example, one such operation can be extracted from the requirement "Each user must have his own private list of bookmarks".
- `emergence`: represents operations that undergo passive transformation. In specific, the state of an object changes without some `actor` forcing it to. For example, the

phrase "the bookmark is re-indexed" indicates that the bookmark undergoes an emergence operation ("re-indexed").

- action: describes an operation performed by an actor on some object. It is the most common operation, including almost all operations that are performed on objects (apart only from ownership). For example, in the requirement "The user must be able to create a bookmark", the term "create" is an action.
- state: indicates an operation that describes the status of an actor. For example, in the phrase "the user is logged in", the state of the actor "user" is "logged in".

The aforementioned owl classes cover effectively the static view of the system, while also storing all information that shall prove useful for the upcoming deliverables of S-CASE.

2.2.2 Ontology Properties

The relations of the ontology are very important since they define the possible interactions between the different classes. In the context of the ontology defined in this Section, we defined a set of properties in order to sufficiently cover all possible interactions.

2.2.2.1 High-Level Ontology Properties

At first, concerning requirements-level, we define the properties shown in Table 1. As shown in that Table, several properties are bidirectional.

Table 1 High-level properties of the static ontology

OWL Class	Property	OWL Class
Project	project_has_requirement	Requirement
Requirement	is_of_project	Project
Requirement	has_compound_requirement	Requirement
Requirement	is_compound_requirement_of	Requirement
Requirement	requirement_consists_of	ThingType, OperationType
ThingType, OperationType	consist_requirement	Requirement

The high-level properties shown in Table 1 cover the interactions among the four main classes of the ontology (Project, Requirement, ThingType, OperationType). In specific, each project can have many different requirements while each requirement can also be compound, i.e. containing other requirements. In addition, each requirement consists of several ThingType and OperationType instances. Furthermore, since OWL allows defining subproperties, we can further refine the requirement_consists_of and consist_requirement properties as shown in Table 2 and Table 3 respectively.

Table 2 Subproperties of the `requirement_consists_of` property

OWL Class	Property	OWL Class
`Requirement`	`requirement_has_concept`	`ThingType`
`ThingType`	`is_concept_of_requirement`	`Requirement`

Table 3 Subproperties of the `consist_requirement` property

OWL Class	Property	OWL Class
`Requirement`	`requirement_has_operation`	`OperationType`
`OperationType`	`is_operation_of_requirement`	`Requirement`

The defined properties are visualized in Figure 2, including only one of the two directions for bidirectional properties for simplicity.

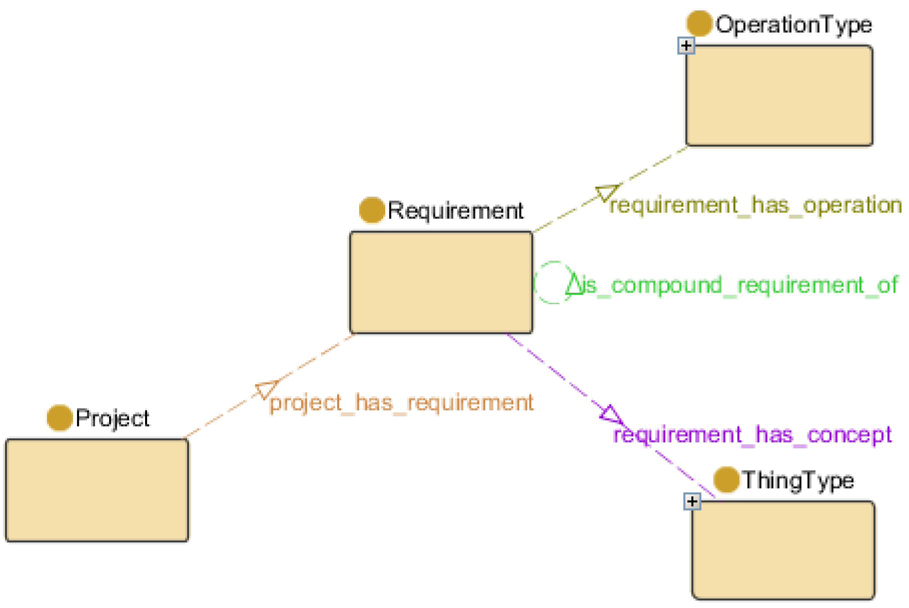


Figure 2 High-level Ontology Properties

2.2.2.2 Low-Level Ontology Properties

With the term "low-level properties" we define the properties that cover the interactions among the different subclasses of `ThingType` and `OperationType`. These properties are given in Table 4.

Table 4 Low-level properties of the ontology

OWL Class	Property	OWL Class
action	acts_on	object, property
object, property	receives_action	action
OperationType	has_actor	actor
actor	is_actor_of	OperationType
object	has_goal	goal
goal	is_goal_of	object
object	has_source	source
source	is_source_of	object
ThingType	has_property	property
property	is_property_of	ThingType
emergence	occurs	object
object	occured_by	emergence
ownership	owns	object
object	owned_by	ownership

As shown in Table 4, several properties are bidirectional. We are able to identify a structure for these properties that is indeed quite similar to the way sentences are structured. In specific, instances of type `actor` are actors of operations, i.e. they are connected with `OperationType` instances via the properties `is_actor_of` and `has_actor`. After that, operations can either connect to objects or not, according to whether they are transitive. Thus, any `action` acts on instances of type `object` or `property`, while `emergence` `occurs` on an `object` and `ownership` is connected with objects via the `owns` and `owned_by` properties. The non-transitive state operation connects only with an `actor` instance (via the properties `is_actor_of` and `has_actor`).

Finally, the composite (or also source-target) nature of the objects is also clearly depicted using properties. Thus, any `object` can have a `source` and/or a `goal`. It connects with the former via the `has_source` and `is_source_of` properties, and with the latter via the `has_goal` and `is_goal_of` properties.

The structure of the subclasses of `ThingType` and `OperationType` along with the properties among them is shown in Figure 3.

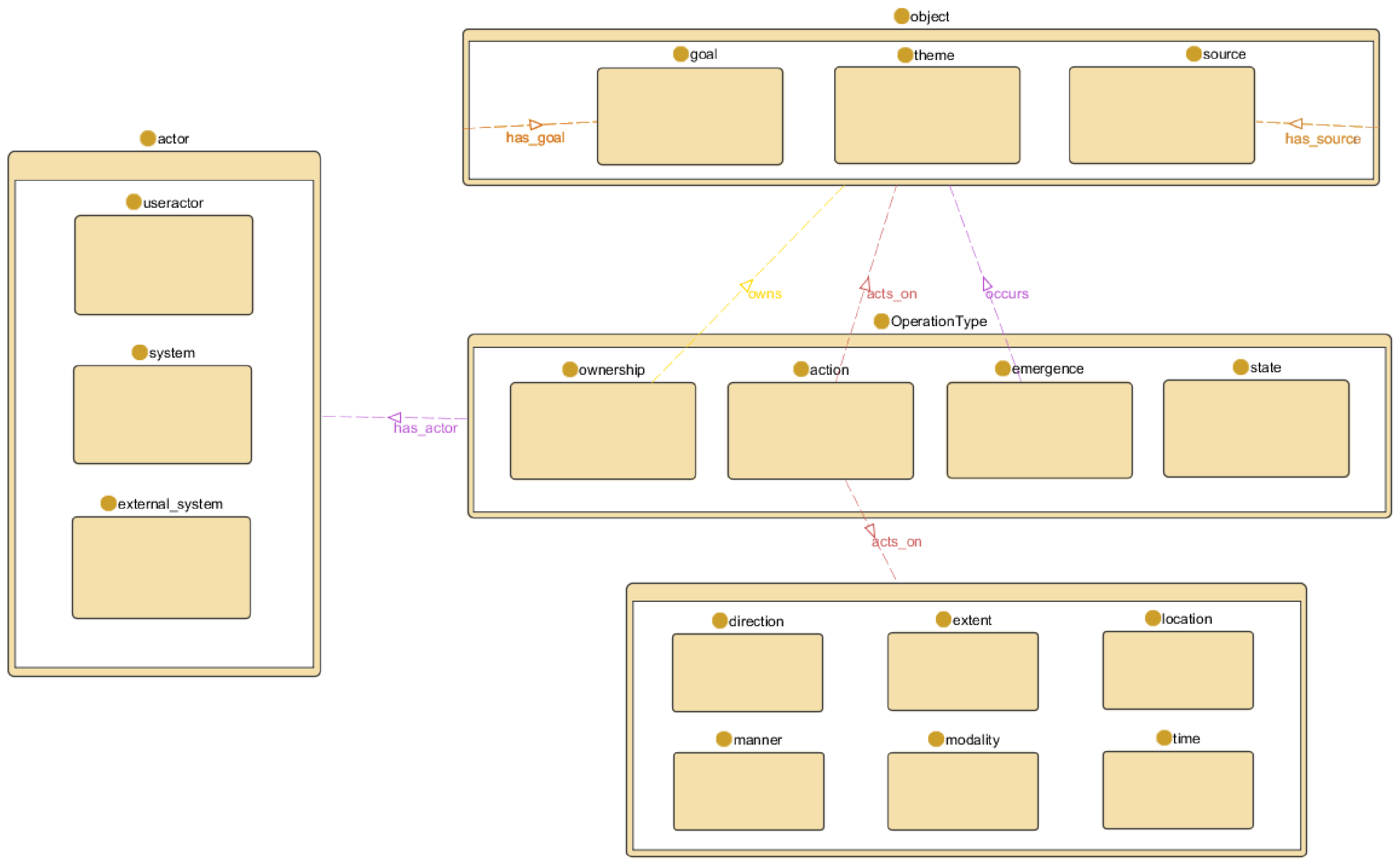


Figure 3 Low-level Ontology Properties

2.3 Example Instances

This Section illustrates the use of the ontology for storing functional requirements. In the following subsections, we present examples of ontology instantiations, including both individual requirements and a whole project.

2.3.1 Individual Instances

An example annotated instance for a functional requirement is shown in Figure 4.

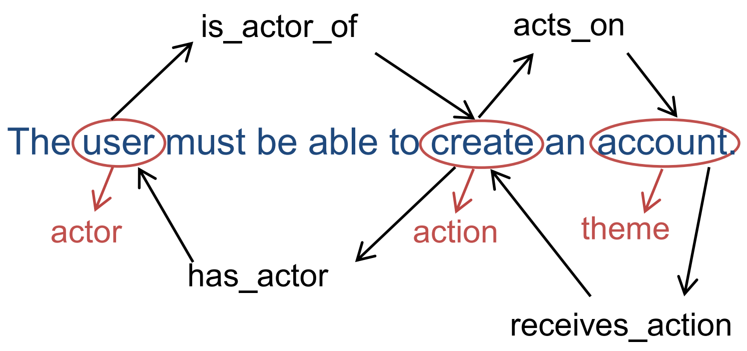


Figure 4 Example annotated instance for a SVO sentence

Though simple, the above example is illustrative of how SVO sentences can be stored in the ontology. Similarly, the ontology can store more complex sentences, including e.g. properties such as the one in Figure 5.

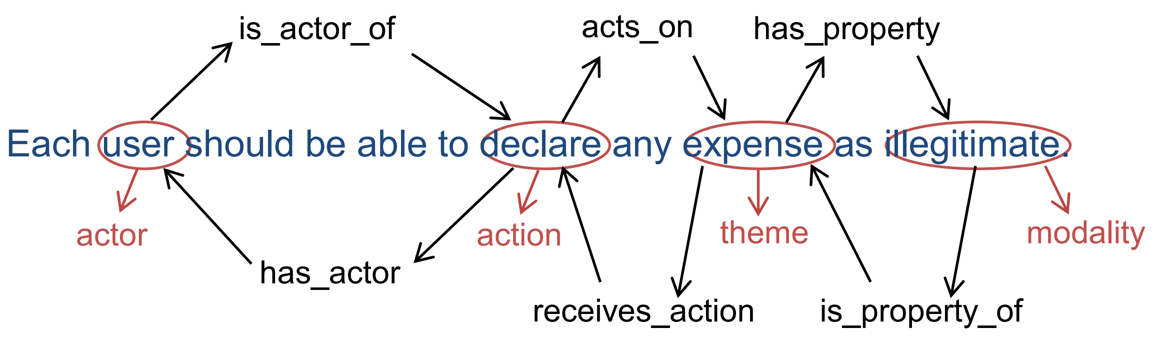


Figure 5 Example annotated instance for a SVO sentence with more modifiers

In the above examples, only low-level owl classes and properties are shown. In the case of a software project, each requirement shall also instantiate the `Requirement` class and its various properties (see Table 2 and Table 3).

Finally, note that this and the next subsection mainly concern the instantiations of the ontology; thus we illustrate how the ontology can handle the instances without referring to how these annotations can be created. The creation of these instances is demonstrated in the next Section with the use of an NLP parser, while manual annotation for training the parser is the topic of Section 4.

2.3.2 Example Project

For illustration purposes, we use the functional requirements of project Restmarks (Project Restmarks, 2014). Restmarks is a social network where each user can share his internet bookmarks. Additionaly, the user can add informative tag to his/her bookmarks, create, modify, or delete existing bookmarks and search for his/her private bookmarks and/or public bookmarks of other users. The functional requirements of Restmarks are given in Figure 6.

FR1. A user must be able to create a user account by providing a username and a password.
FR2. A user must be able to login to his/her account by providing his/her username and password.
FR3. A user that is logged in to his/her account must be able to update his password.
FR4. A logged in user must be able to add a new bookmark to his/her account.
FR5. A logged in user must be able to retrieve any bookmark from his/her account.
FR6. A logged in user must be able to delete any bookmark from his/her account.
FR7. A logged in user must be able to update any bookmark from his/her account.
FR8. A logged in user must be able to mark his/her bookmarks as public or private.
FR9. A logged in user must be able to add tags to his/her bookmarks.
FR10. Any user must be able to retrieve the public bookmarks of any RESTMARKS's community user.
FR11. Any user must be able to search by tag the public bookmarks of a specific RESTMARKS's user.
FR12. Any user must be able to search by tag the public bookmarks of all RESTMARKS users.
FR13. A logged in user, must be able to search by tag his/her private bookmarks as well.

Figure 6 Functional requirements of project Restmarks

Given the requirements of the project, one can construct the ontology instance shown in Figure 7. As shown in that Figure, classes `theme`, `action`, and `property` (`modality` and `manner`) are among the most used ones. Requirements typically also have an `actor`, which however is the same for several requirements (e.g. "user").

One can make several observations based on the instantiation of the ontology shown in Figure 7 with respect to the requirements of Figure 6. Note for example the "user" instances. There is a `useractor` instance "user" and a `modality` instance "user_1". Both instances however are correctly stored, since the `useractor` instance refers to the actor of the requirements (which is in this case the same in all 13 requirements), whereas instance "user_1" refers to the word "user" which is found in the 1st requirement of the project as an adjective of the noun "account" (see Figure 6).

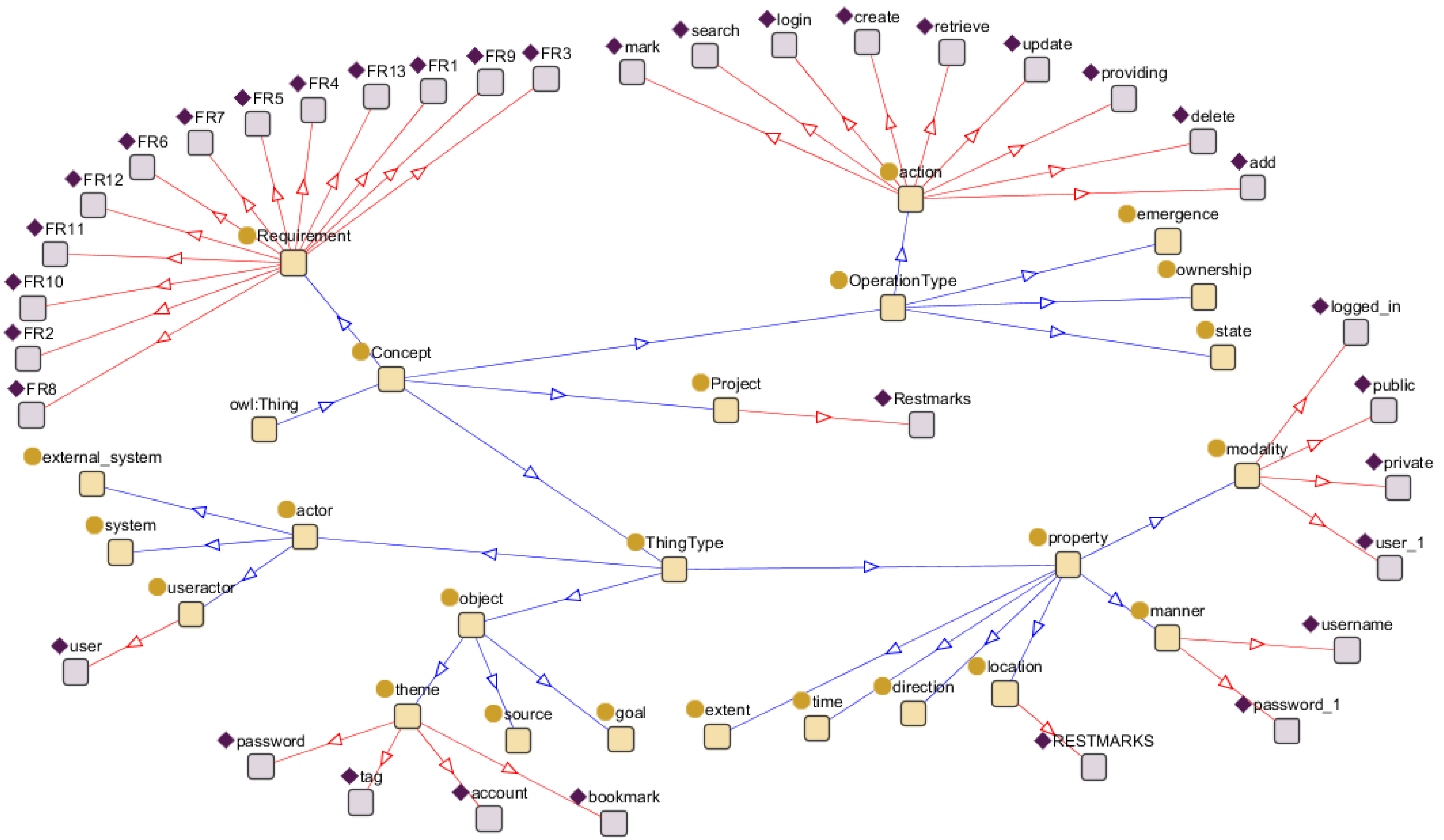


Figure 7 Example ontology instance for project Restmarks

Note also that several terms may be instances of `object` and `property` at the same time. This is also expected since they can be modeled either as objects that can be deleted, updated, etc. or as properties (similar to sub-objects) of other objects. For example, the term "password" appears in three requirements of the project. In the 3rd requirement it is an object, while in the 1st and the 2nd requirements it is better modeled as a property of "account".

Finally, in Table 5, the low-level properties of requirement FR4 of Restmarks are presented. The properties once again imply a SVO structure (`user–add–bookmark`), while the `goal` instance is also used in order to model the fact that an `account` is the goal of `bookmark`s (i.e. the user account contains his/her bookmarks)

Table 5 Low-level properties for the ontology instances of the FR4 of Restmarks

OWL Individual	Property	OWL Individual
`user`	`is_actor_of`	`add`
`add`	`has_actor`	`user`
`add`	`acts_on`	`bookmark`
`bookmark`	`receives_action`	`add`
`bookmark`	`has_goal`	`account`
`account`	`is_goal_of`	`bookmark`

3 Module for Extracting Software Artefacts

The module for extracting software artefacts from text is based on the concept ontology described in Section 2. That is, the module processes text and automatically detects fragments that instantiate concepts and relations defined in the ontology and maps them to a corresponding meaning representation. In practice, this *parsing* task involves several steps: first, instances of concepts need to be identified and then mapped to the correct class and, second, relations between instances of concepts need to be identified and labelled accordingly.

While there is little previous work on analysing software requirements using semantic parsing techniques, various methods have been proposed for related tasks in natural language processing. Early work relied on custom-built syntactic parsers and simple rules for mapping grammatical relations to logical symbols (Nanduri and Rugaber, 1995). However, building special-purpose grammars for specific domains is labour-intensive and scales. From both an engineering and a linguistic perspective, it is more appealing to start from an existing broad-coverage grammar and modify it to address the relevant domain. One such approach would be to couple semantic and syntactic analysis through a transparent interface as proposed, for example, in the combinatory categorial grammar formalism (Steedman, 2000). An alternative, more conventional approach, is to perform syntactic analysis first and then apply semantic role labelling (SRL) techniques that assign thematic relations (i.e., *who* did *what* to *whom*) to words-spans based on syntactic structure (Gildea and Jurafsky, 2002).

For this module, we implemented a parsing pipeline based on previous work in semantic role labelling (cf. Figure 8). This choice was motivated by an initial analysis of a small set of software requirements in which we tested several previous methods and found semantic role labelling techniques to generally provide the best off-the-shelf results. Other approaches turned out to generalize less well or did not provide robust output, leading to higher error rates and coverage gaps. In our implementation, we adapt semantic role labelling techniques from purely linguistically motivated relations to directly utilize the concepts and relations defined in the S-CASE ontology.

The following sub-sections describe our implementation in more detail. In Section 3.1, we introduce the preprocessing pipeline that we apply to compute a syntactic analysis for each requirement expressed as a sentence in English. Section 3.2 describes the semantic analysis modules that we implemented to map words and constituents in a sentence to instances of concepts and relations from the ontology. The mapping is based on statistical models that are trained on annotated data (cf. Section 4). We define the *features* and learning techniques applied to train each statistical model in Section 3.3 and 3.4, respectively. Throughout this section, we provide an example analysis for the following sentences:

(a) "The user must be able to create an account."
(b) "Any user must be able to search by tag the public bookmarks of all RESTMARKS users."

A prototype implementation of the module for extracting software artefacts from text is provided in Appendix A.

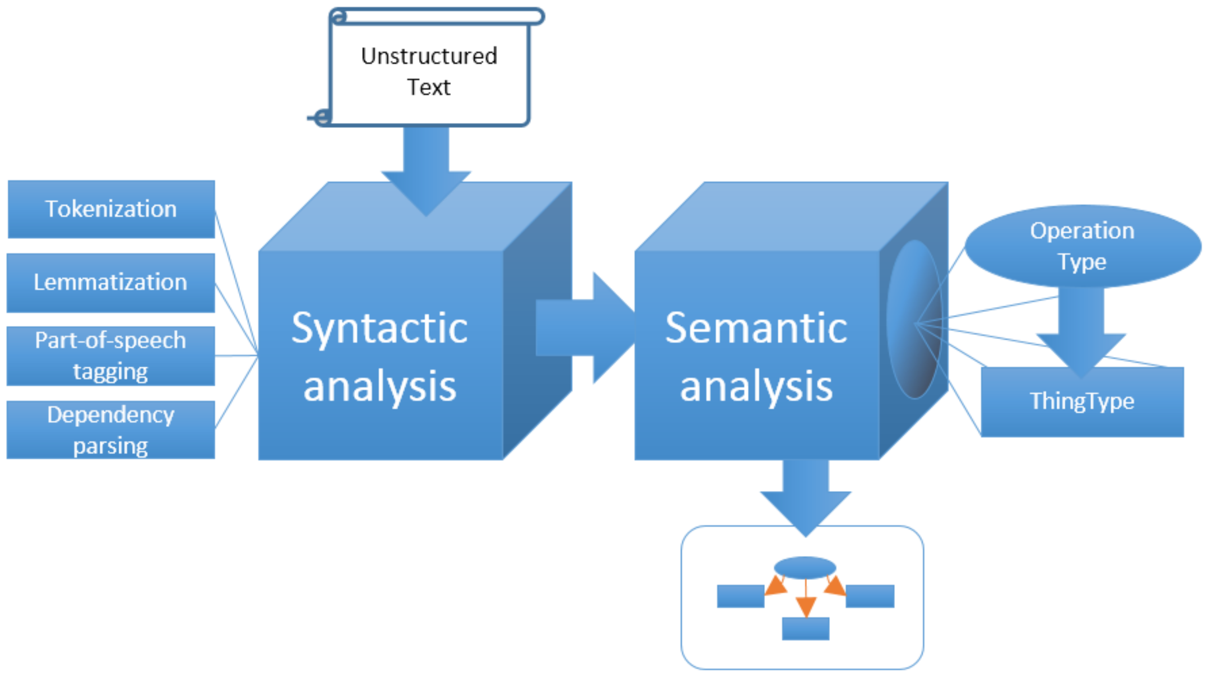


Figure 8: Illustration of our processing pipeline.

3.1 Syntactic Analysis

The syntactic analysis stage of our pipeline architecture performs the following syntactic analyses: *tokenization*, *part-of-speech tagging*, *lemmatization* and *dependency parsing*. Given an input sentence, this means that the pipeline separates the sentence into word tokens, identifies the grammatical category of each word (e.g., `user' → *noun,* `create' → *verb*) and determines their uninflected base forms (e.g., `users' → `user'). Finally, the pipeline identifies the grammatical relations that hold between two words (e.g., <`user',`must'> → *subject-of*, <`create',`account'> → *object-of*).

For all syntactic analysis steps, we rely on components of a readily available system called mate-tools (Björkelund et al., 2010; Bohnet, 2010). This choice is based on three criteria: (1) the system achieves state-of-the-art performance on a benchmark data set for syntactic analysis (Hajič et al., 2009), (2) the output of the syntactic analysis steps has successfully been used as input for related semantic parsing problems, and (3) the system is fast and robust, meaning that it can be integrated efficiently into the S-CASE platform.

3.2 Semantic Analysis

The semantic analysis components that we implemented for extracting software artefacts from text consists of four main components, which we describe in more detail in the following sub-sections. The sub-tasks accomplished by the components are (1) identifying instances of `OperationType`; (2) allocating these to the correct sub-class; (3) identifying instances of `ThingType` and (4) determining their relationships to instances of `OperationType`.

3.2.1 Identifying instances of OperationType

The component for identifying instances of OperationType detects words in a text that express actions, ownerships, emergence processes and states of systems or entities (e.g., `create’, `search’). The component considers each *verb* and each *noun* in a sentence as a potential instance of the OperationType concept and performs binary classification based on lexical semantic and syntactic properties of each candidate.

3.2.2 Classifying instances of OperationType

The component for classifying instances of OperationType determines which subtype is applicable to each instance determined by the identification component. That is, for each verb and noun in a sentence classified as a potential instance of OperationType, this component determines whether the specific case expresses an instance of action, ownership, emergence or status (e.g., `create’ → action, `search’ → action). As in the previous component, lexical semantic and syntactic properties are exploited to perform classification.

3.2.3 Identifying instances of ThingType

The component for identifying instances of ThingType detects words and phrases in a text that refer to instances of properties and participants, as defined in the ontology (cf. Section 2). The main goal of this component is to recognize instances that are related in a meaningful way to instances of OperationType or to other instances of ThingType. Accordingly, the component takes as input pairs of potential instances and performs binary decisions that indicate whether they are related or not. Only candidate instances of ThingType that are found to be related to another (potential) instance are identified as such. In example (a), both `the user’ and `an account’ are instances of ThingType that are recognized as related to the action expressed by the word `create’. In example (b), instances related to `search’ are: `any user’, `by tag’ and `the public bookmarks of all RESTMARKS users’, with `of all RESTMARKS users’ being itself a property related to the instance of ThingType expressed by the phrase `the public bookmarks’.

3.2.4 Classifying instances of ThingType

The component for classifying instances of ThingType determines suitable subtypes for each instance of ThingType determined by the corresponding identification module. Naturally, entities can be involved as properties and participants in different relations and hence multiple subtypes can apply to a single identified instance. To represent different aspects of an instance with respect to each relation separately, the component performs classification on pairs of related instances, similar to the (binary) classification in the previous component (e.g., <`the user’, `create’> → <actor, action>). In this classification step, lexical semantic and syntactic properties are complemented by additional characteristics that hold between the linguistic expressions that refer to the considered instances (e.g. their order in text).

3.3 Features

As indicated in the description of each component, the module for extracting software artefacts makes use of a wide range of linguistic properties to identify and classify instances of `OperationType` and `ThingType`. In practice, each decision is performed by a statistical model that uses linguistic properties as features, for which appropriate features weights are determined based on annotated *training data*. We next provide a full list of features that are used in the statistical models. Details on the learning process can be found in Section 3.4.

3.3.1 Binary Features

The features defined in this sub-section are binary indicator features that take values of 1 (applicable) and 0 (not applicable). Each item listed in this sub-section refers to a set of binary features, each of which is automatically generated from the data using a simple feature template. For example, the template “word form” refers to a set of features with each representing one particular word form. When processing new input, a feature value is set to 1 if and only if the indicated property holds true for the word to be classified.

OperationType features. We use the following sets of features on nouns and verbs to identify whether they express an instance of an `OperationType` and, if so, to classify which sub-type is expressed. To avoid ambiguity, we refer to the specific word to be classified as *the predicate*.

- Part-of-speech assigned to the predicate (e.g., *noun*, *verb*)
- Dependency relation from the predicate to its head word in the syntactic tree, if any (e.g., *subject*, *object*)
- Word form of the head word, if any
- Part-of-speech assigned to the head word, if any
- Set of dependency relations to children of the predicate, if any (e.g., *{subject,object}*)
- Each single dependency relation to any child of the predicate (e.g., *subject*, *object*)
- Word form of each single child of the predicate
- Part-of-speech assigned to each single child of the predicate

ThingType features. We define a similar set of features to identify instances of `ThingType` and to determine their relation to other instances of concepts in the ontology. We refer to candidate instances as *arguments* and call previously recognized instances *predicates*, reflecting the fact that instances of `ThingType` are typically in a directed relation to a previously recognized concept instance. Note though that predicates can themselves be arguments of other predicates, hence features are always computed for a specific pair. In case a potential argument is expressed by a phrase of more than one word, we represent the corresponding word span by the head word of the phrase according to the syntactic tree of the sentence (e.g., \`the user’ → \`user’). The complete dependency trees computed by the preprocessing pipeline for examples (a) and (b) are shown in Figure 9 and in Figure 10, respectively.

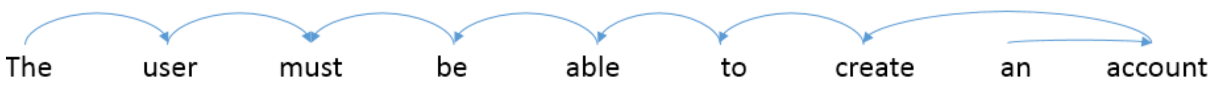


Figure 9: Dependency tree computed by the preprocessing pipeline for example (a).

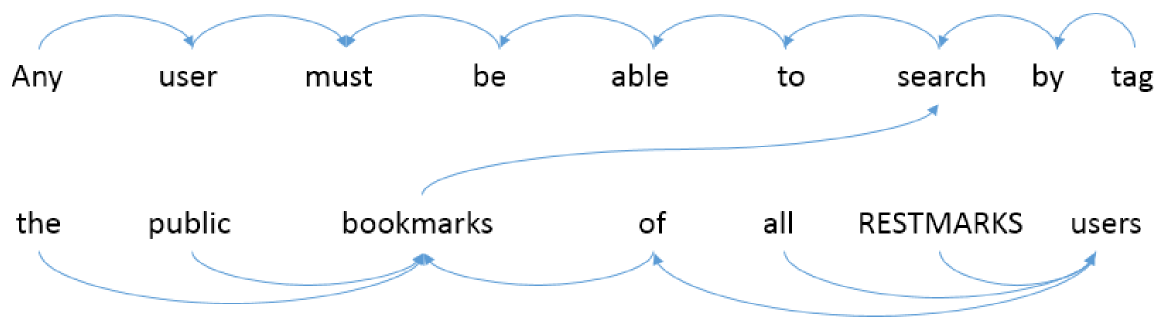


Figure 10: Dependency tree computed by the preprocessing pipeline for example (b).

The following feature templates are applied on the preprocessed sentence to derive indicator features:

- Word form of the predicate (e.g., \`create', \`search')
- Part-of-speech assigned to the predicate (e.g., *verb*)
- Lemmatized word form of the predicate (e.g., *create, search*)
- Concept type assigned to the predicate (e.g., *action*)
- Word form of the head of the predicate (e.g., \`to')
- Part-of-speech assigned to the head word of the predicate (e.g., *preposition*)
- Each single dependency relation to any child of the word (e.g., *object, adverbial complement*)
- Word form of each single child of the word (e.g., \`account', \`by', \`bookmarks')
- Part-of-speech assigned to each single child of the word (e.g., *noun, preposition*)
- Word form of the argument (e.g., \`user', \`account', \`bookmarks')
- Part-of-speech assigned to the argument (e.g., *noun*)
- Dependency relation of the argument to its head word in the syntactic tree, if any (e.g., *subject*, *object*)
- Dependency path from the argument to the predicate according to the syntactic tree (e.g., *<subject,object>*),
- List of part-of-speech tags assigned to all words in the dependency path (e.g., *{noun,verb}*)
- Relative position of the argument with respect to the predicate (e.g., *left, right*)
- Word form of the left-most dependent of the argument according to the syntactic tree (e.g., \`the')
- Part-of-speech assigned to the left-most dependent of the argument (e.g., *determiner*)
- Word form of the right-most dependent of the argument (e.g., \`of')
- Part-of-speech assigned to the right-most dependent of the argument (e.g., *preposition*)
- Word form of the next right sibling of the argument in the dependency tree (e.g., \`be')
- Part-of-speech assigned to the next right sibling of the argument (e.g., *verb*)
- Word form of the next left sibling of the argument according to the syntactic tree (e.g., \`by')
- Part-of-speech assigned to the next left sibling of the argument (e.g., *preposition*)

3.3.2 Continuous Features

In addition to the indicator features described in the previous sub-section, we define a small set of continuous features that are applicable to any word involved in a classification decision. The motivation for this additional feature set lies in the fact that indicator features can be too sparse and hence too specific to provide robust generalization for semantic parsing. To overcome the resulting gap in coverage, we specifically use as additional features distributional word representations that are based on word-context co-occurrences and can be computed over large amounts of unlabelled text. Following the *Distributional Hypothesis* (Harris, 1954)—popularly known as "a word is characterized by the company it keeps" (Firth, 1957), words that are similar in meaning also have similar distributional representations and can hence be used in place of one another, for example, if there is no corresponding indicator feature available.

In our classification components, we apply a set of publicly available word representations.[4] We experimented with different settings on an out-of-domain data set and found word representations to perform best that were learned from unannotated text using a neural language model (Bengio et al., 2006) and 50-dimensional vector representations. In neural language models, representations are trained for each word type by optimizing an objective function, in which each word in a text is predicted given the (representations of the) *n* surrounding words. The representations hence reflect syntactic and semantic properties that can be derived from the typical contexts in which a word appears. As examples, the following representations are learned for the words `user', `account' and `tag':

```
user     := [-0.37, -0.20, -0.46, -0.08, 0.53, -0.24, ..., 0.07]
account  := [-0.48,  0.06,  0.03,  0.23, 0.25,  0.06, ...,-0.02]
tag      := [-0.48,  0.00,  0.00, -0.02, 0.31, -0.22, ...,-0.35]
```

Interpreting these representations as geometrical vectors, each one points into a different direction in a vector space. The latter two vectors are, however, closer to each other than to the first one. In our semantic analysis modules, we utilize the directions of vectors for classification by looking up the representations of predicates and arguments involved in a classification decision and by applying each component of a vector as an additional feature.

3.4 Learning

As discussed in Section 3.3, the module for extracting software artefacts consists of several semantic analysis modules that rely on a range of linguistic properties, which can be extracted from text using various preprocessing techniques. We rely on each property as a feature for statistical classification and make sure that classification decisions are affected in a suitable manner by learning appropriate feature weights from annotated data.

For each component of the module discussed in this document, we use the logistic regression classifier implemented in the LIBLINEAR toolkit (Fan et al., 2008). The underlying

[4] http://metaoptimize.com/projects/wordreprs/

statistical computation in this toolkit is performed by iteratively optimizing the feature weights **w** given feature values **x** and a binary classification label *y* following equation (1):

$$\min_{\mathbf{w}} \log\left(1+e^{-y\mathbf{w}\mathbf{x}}\right)+0.5\mathbf{w}^{T}\mathbf{w} \qquad (1)$$

The first part of equation (1) is the logistic loss, which is used to minimize the feature weights **w** such that the output of the logistic function applied to **wx** is close to *1* iff *y*=1. The second part of equation (1) is a convex regularization constraint that ensures feature weights stay close to zero, in order to avoid *overfitting* to the training data ($\mathbf{w}^T$ indicates the transpose of vector **w**). As input for learning, we use an annotated training data set, in which words in text are directly related to concepts and relations from the ontology (cf. Section 2 for details). We apply our preprocessing components described in Section 3.1 to extract feature values and derive class labels from annotated concepts and relations. In the identification modules (cf. Section 3.2.1 and Section 3.2.3), we use the class label *1* to indicate that a word expresses an instance of an ontology concept (otherwise -*1*). In the case of multi-way classification decisions (cf. Section 3.2.2 and Section 3.2.4), a one-vs.-all model is learned for each concept in the ontology.

4 Collection and Annotation of Software Requirements

Training an NLP parser is a hard task. It requires collecting and annotating appropriate datasets so that the parser can distinguish the ontology class of the instances. In this Section, we initially describe our efforts on collecting datasets. After that, we analyze the annotation scheme followed and refer to an annotation tool designed and implemented specifically in order to annotate functional requirements.

4.1 Data Collection

Since software requirements can drastically differ in quality, style and granularity, we created a highly diverse dataset including requirements documents from various domains.

A large part of the collected documents came from a software development course organized jointly by several European universities (Distributed Software Development, 2013). The student projects of this course focused on several different areas, such as embedded systems, virtual reality and web applications. In total, we collected 270 requirements from over 100 student projects.

Additionally, since the usage scenario of S-CASE involves prototyping RESTful projects, we collected the functional requirements of the RESTAPPS (Project Restmarks, 2014; inter alia). These are probably most typical of the type of requirements expected concerning the projects that shall be created using S-CASE. Furthermore, any requirements from pilot cases shall certainly be useful. Except for the GiftCase prototype, however, requirements were not available at the time of the writing of this deliverable. From RESTAPPS and GiftCase, we acquired 26 and 29 requirements, respectively. Together with the data from student projects, our collection at this point amounts to 325 requirements, with an average length of 12 words and a total vocabulary size of 765 word types.

Finally, data collection involved also requirements documents from past projects (S-CASE wiki, 2014). These were collected from all partners of S-CASE and involve requirements from other EU-funded projects. Although these requirements may prove useful for diversity, they are generally not close enough to the scenario of S-CASE since they are too generic.

4.2 Annotation Scheme

Upon creating a dataset consisting of software requirements, the next step is to annotate these requirements in order to train the parser. The main issue here lies in deciding how complex these annotations should be. In specific, an annotation scheme that is very close to the ontology classes described in Section 2 would be ideal for training the parser (since this is the final desired result). However, such a scheme would be very difficult for annotators without sufficient background knowledge.

As a result, we propose a multi-step annotation scheme in which decisions in one iteration are further refined in later iterations. By adopting the class hierarchy introduced in Section 2, we can naturally divide each annotation iteration according to a level in the ontology. This means that in the first iteration, we ask annotators to simply mark all instances of `actor`, `object`, `OperationType`, and `property` that are explicitly expressed in a given requirement. After that, further refinements can be made (by more experienced annotators) in order to select more specific subclasses for each instance. Thus, we add one layer of sophistication from the class hierarchy in each iteration, resulting in step-wise refinements. In the final iteration, we can also add implicit but inferable cases of relations between instances of concepts (e.g. in the phrase “a user can delete his/her account” involves not only an action performed on “account” but also ownership of the “account” by the “user”).

Consider the example of Figure 11:

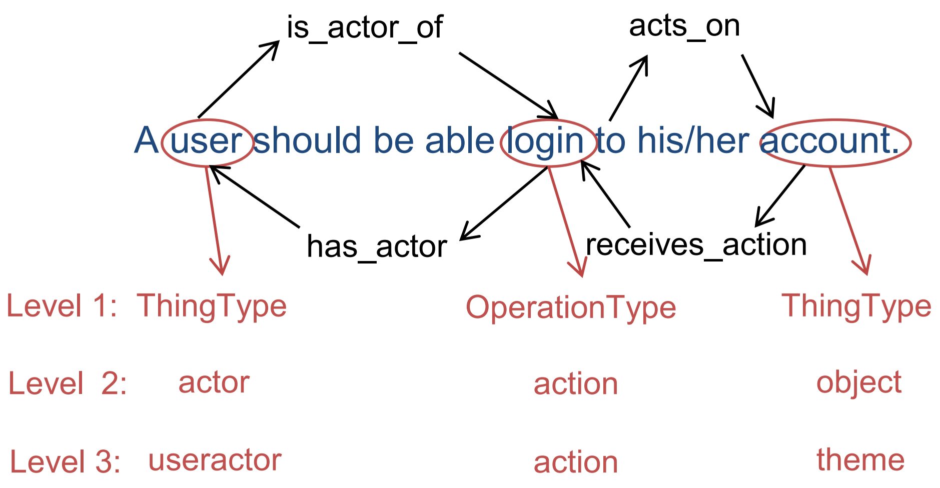


Figure 11 Example annotated instance using the hierarchical annotation scheme

In this sentence, the first iteration would include annotating the “user” and the “account” as instances of `ThingType` and the “login” as an `OperationType` and the “account” as an `object`. The second iteration would include annotating the “user” as an `actor`, the “login” as an `action` and the “account” as an `object`. After that, the next iteration would involve specifying the “user” as a `useractor`, and the “account” as a `theme`. Finally, in this example we could also add one more iteration where we would specify “account” as an object `owned_by` “user”. This relation is not explicitly given in this sentence, however it is correct.

4.3 Annotation Tool

As noted in the previous subsections, annotating is usually too hard of a task for inexperienced users. In our case, training the parser would involve providing it with large annotated datasets. As a result, we had to create an annotation tool that implements a simple and easy to use approach to annotation. It concerns the first levels of the annotation scheme defined in the previous subsection, i.e. asking users to define actors, actions, objects and properties.

We named our tool “S-CASE Requirements Annotation Tool”. We provide an introduction of our tool here, including a comprehensive example of its usage, without however fully presenting the development of the tool. The full documentation of the tool shall be included in the deliverable 5.2 (Tools for developers) of S-CASE.

4.3.1 Usage of the Annotation Tool

The S-CASE Requirements Annotation Tool is a web platform that allows users to create an account, import one or more of their projects and annotate them. As mentioned above, the tool allows specifying terms (or phrases) as one of the following entities:

- Actor
- Action
- Object
- Property

Concerning relations between these terms, the following relations are available:

- IsActorOf, which is declared from Actor to Action
- ActsOn, which is declared from Action to Object or from Action to Property
- HasProperty, which is declared from Actor to Property, or from Object to Property, or from Property to Property

Notice that we refrain from declaring also the opposite relations (e.g. HasActor) in order to keep the tool as simple as possible. Thus, the tool presents a very simple task to the user; there are only 4 entities and 3 relations, while all of them are quite close to the definitions of the English language. In specific, the triple Actor-Action-Object is actually quite similar to SVO, while Property represents mostly modifiers (adjectives, phrases, etc.). Finally, the tool offers the option of automatically annotating software projects in order to facilitate the process of manual annotation.

Mapping the annotated terms to the ontology is quite straightforward, yet not trivial. The mapping is given in Table 6.

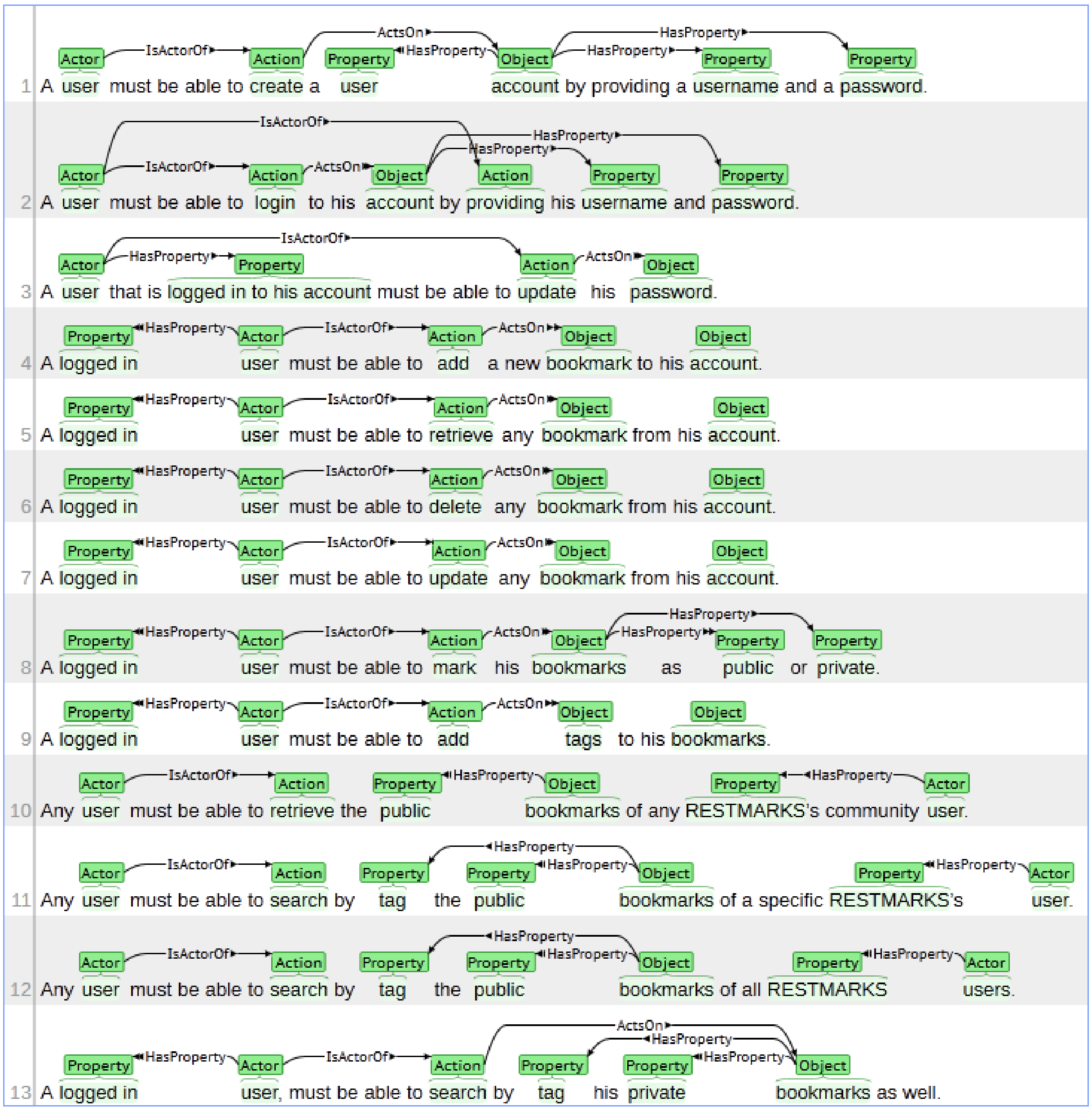
Table 6 Mapping of S-CASE Requirements Annotation Tool entities and relations to the ontology

Annotation Tool Entities and Relations	**OWL Class or Property**
Actor	`actor`
Action	`OperationType`
Object	`object`
Property	`property`
IsActorOf	`is_actor_of, has_actor`
ActsOn	`acts_on, receives_action`
HasProperty	`has_property, is_property_of`

Note especially how Action is not mapped to `action`, but rather to `OperationType`. This is due to the user not specifying whether the defined operation is indeed an `action`, or one of the other three subclasses of `OperationType`. Additionally, transitive relations, such as IsActorOf, are mapped to ontology properties also defining the opposite properties, i.e. for IsActorOf both `is_actor_of` and `has_actor` are defined. Finally, since the identifiers of the requirements as well as the project are known, the ontology classes `project` and `requirement` are also instantiated, including all the respective high-level properties, e.g. project_has_requirement, requirement_has_concept, etc. (see Table 1, Table 2, and Table 3).

4.3.2 Example Annotated Project using the Annotation Tool

We provide here an example of using the annotation tool in order to clarify its cause and illustrate how it can help create annotated instances out of software project requirements. For this example, we use the functional requirements of project Restmarks (see Figure 6). The annotated requirements are shown in Figure 12.



1 A user must be able to create a user account by providing a username and a password.
2 A user must be able to login to his account by providing his username and password.
3 A user that is logged in to his account must be able to update his password.
4 A logged in user must be able to add a new bookmark to his account.
5 A logged in user must be able to retrieve any bookmark from his account.
6 A logged in user must be able to delete any bookmark from his account.
7 A logged in user must be able to update any bookmark from his account.
8 A logged in user must be able to mark his bookmarks as public or private.
9 A logged in user must be able to add tags to his bookmarks.
10 Any user must be able to retrieve the public bookmarks of any RESTMARKS's community user.
11 Any user must be able to search by tag the public bookmarks of a specific RESTMARKS's user.
12 Any user must be able to search by tag the public bookmarks of all RESTMARKS users.
13 A logged in user, must be able to search by tag his private bookmarks as well.

Figure 12 Annotated requirements of project Restmarks

As shown in Figure 12, the annotations are comprehensive; any user with no experience or training shall be able to correctly identify and label the appropriate entities and relations.

Upon annotating a project, the tool can export the annotations in different forms, including the owl and ttl ontology forms. Thus, for the Restmarks project we visualize the owl that is provided by the annotation tool in Figure 13.

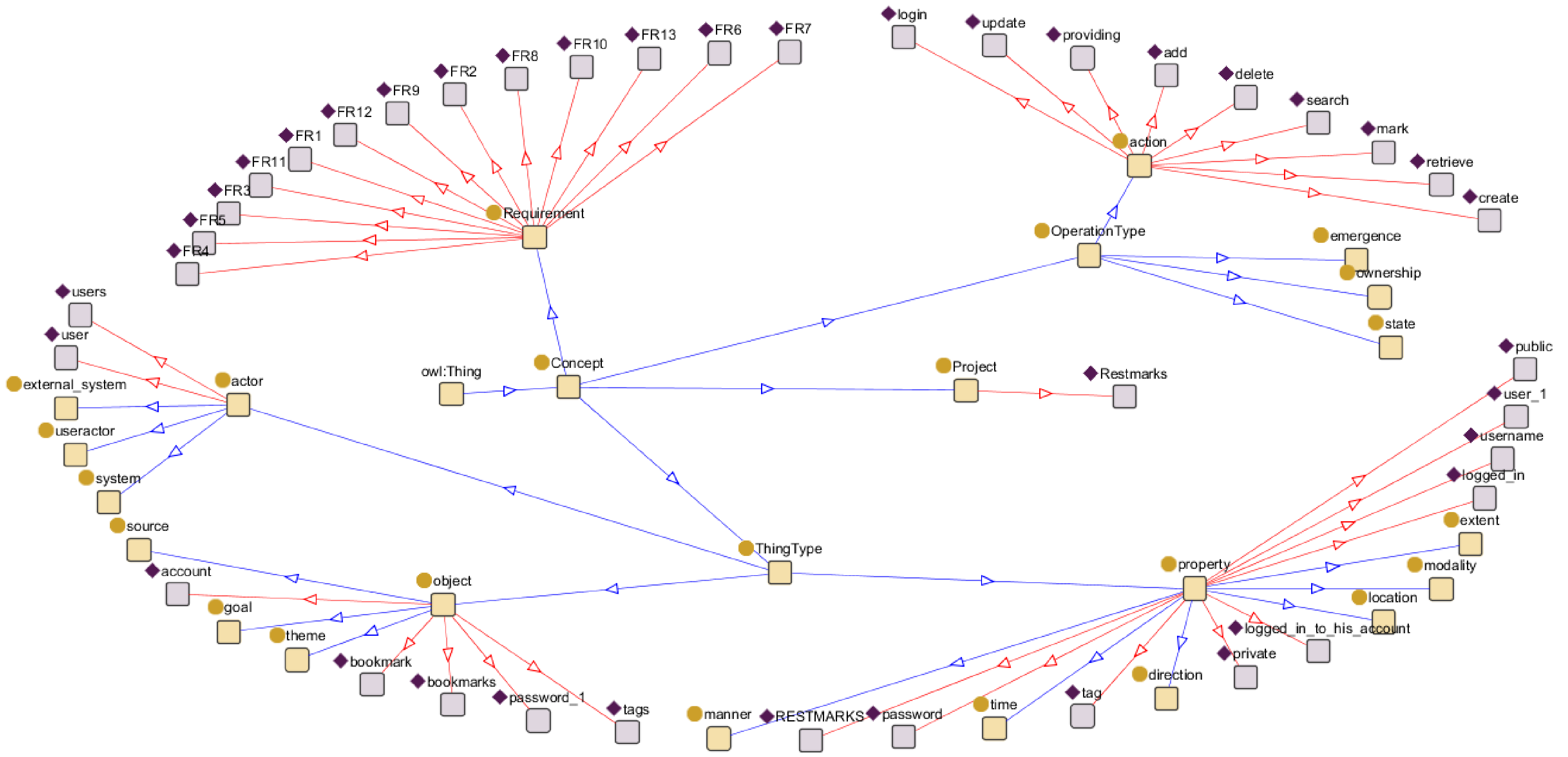


Figure 13 Ontology instance for project Restmarks, annotated using the S-CASE Requirements Annotation Tool

One can spot several differences between the initial ontology instantiation provided in Figure 13, and the finalized one given in Figure 7 for the same project. The annotation scheme of subsection 4.2 is now clear. For example, terms such as “bookmark” or “tag” are instances of class `object` in Figure 13 and can be further refined as instances of `theme` in Figure 7. This also happens with instances of class `actor` and the subclass `useractor`, as well as the class `property` and its various subclasses. Operations are generally instances of `action` since this is the most usual subclass of `OperationType`**.** Note that the parser can use synonym and type lexicons in order to find similar terms such as “bookmark” and “bookmarks” and keep one of the two.

Concerning the properties of the ontology individuals, the two ontologies are once again similar. For example, the low-level properties for the individuals of FR4 of Restmarks are shown in Table 7. Comparing this table with Table 5, one can clearly see that the only difference is the absence of the `has_goal` and `is_goal_of` properties between the `bookmark` and `account` instances in Table 7. Since this information is not given by the annotation tool (see Figure 12) it has to be provided manually in the next level of the annotation scheme.

Table 7 Low-level properties for the ontology instances of the FR4 of Restmarks

OWL Individual	**Property**	**OWL Individual**
`User`	`is_actor_of`	`add`
`add`	`has_actor`	`user`
`add`	`acts_on`	`bookmark`
`bookmark`	`receives_action`	`add`

Finally, using ontology software such as Protégé (Protégé, 2014), one can easily perform the required operations of the hierarchical annotation scheme in order to construct the ontology instance of Figure 7 given the initial instance of Figure 13. Since the relations are defined in the first level of the hierarchy, assigning different classes to certain individual instances is simple.

5 Conclusions

The work discussed in this document summarizes our progress to date on the task of recognizing and providing a formal semantics for informally expressed software requirements in unstructured and semi-structured text (T3.1). We achieved the following objectives related to this task:

- We collected an initial corpus to analyse and determine linguistic and conceptual characteristics involved in software requirements.
- Based on our analyses, we devised an ontology that defines the concepts and relations needed to formally represent static functionalities of a software system.
- We tested various methodologies for parsing functional requirement semantically and implemented a pipeline architecture that can deal with the domain-specific properties of the analysed input.
- We devised an annotation scheme and implemented a corresponding tool for S-CASE users to provide and revise mappings from text fragments to ontology concepts and relations manually.

Our goal is to continuously improve the module for extracting software artefacts from text throughout the remainder of the project. Towards this goal, we are currently using the implemented annotation tool to collect additional annotations that will help us retrain our parsing pipeline to achieve better performance. We are further exploring the possibility of extending our parser to match the objectives of sub-sequent tasks in WP3, including its application in the question answer system to be developed in Task 3.4. For validation purposes, we are planning to evaluate our approach intrinsically—by comparing the parser's output to manual annotations on a held-out data set—and extrinsically, within the question-answer scenario.

References

Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. 2006. Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137—186. Springer Berlin Heidelberg.

Anders Björkelund, Bernd Bohnet, Love Hafdell, and Pierre Nugues. 2010. A High-Performance Syntactic and Semantic Dependency Parser. In *COLING 2010: Demonstration Volume*, pages 33—36. Association for Computational Linguistics.

Bernd Bohnet. 2010. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics*, pages 89—97. Association for Computational Linguistics.

Rong-En Fan, Kai-Wie Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research 9:1871—1874*. Microtome Publishing.

John R. Firth. 1957. A synopsis of linguistic theory 1930—1955. In *Studies in Linguistic Analysis,* Philological Society, Oxford; reprinted in Palmer, F., (ed. 1968), *Selected Papers of J.R. Firth*. Longman, Harlow.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Stranák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. 2009. The CoNLL-2009 shared task: Syntactic and semantic dependencies in multiple languages. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task*, pages 1—8. Association for Computational Linguistics.

Zellig S. Harris. 1954. Distributional structure. *Word* 10(2/3):146—162.

Daniel Gildea and Daniel Jurafsky. 2002. Automatic Labeling of Semantic Roles. *Computational Linguistics* 28(3):245—288. MIT Press.

Sastry Nanduri and Spencer Rugaber. 1995. Requirements validation via automated natural language parsing. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, volume 3, pages 362—368.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press.

S. Palmer, Martha, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics* 31(1): 71—106.

Project Restmarks, RESTAPPS, S-CASE Consortium, 2014.

Distributed Software Development, Joint Course, Mälardalen University (MdH), School of Innovation, Design and Engineering (IDT), Sweden, University of Zagreb, Faculty of Electrical Engineering and Computing (FER), Croatia, Politecnico di Milano, (POLIMI), Information Engineering School, Italy, 2013, available online: http://www.fer.unizg.hr/rasip/dsd

S-CASE wiki, 2014, available online: http://wiki.scasefp7.com

Protégé, Stanford Center for Biomedical Informatics Research (BMIR), Stanford University School of Medicine, 2014, available online: http://protege.stanford.edu

A. NLP Parsing Prototype

Our parsing prototype for extracting software artefacts from text is publicly available under the following URL:

```
http://www.scasefp7.eu/asset/semantic-parsing-prototype/
```

The ZIP archive contains a standalone version of our parser, including the preprocessing pipeline and semantic analysis modules described in Section 3, implemented as platform-independent JAVA code.

The standalone version of the parser takes text files as input that contain one sentence per line and automatically induces semantic annotations for each input sentence. To run the parser, the following command needs to be executed on the command line:

```
sh parse.sh <FILENAME>
```

Corresponding output, represented using the concepts and relations introduced in Section 2, will be written to a file with the same filename as the input using the file extension `.ann`. The written file contains annotations in a tabular structure, describing mappings from character positions to ontology instances as well as relations between instances.