Software Requirements as an Application Domain for Natural Language Processing

Language Resources and Evaluation

Software Requirements as an Application Domain for Natural Language Processing

Themistoklis Diamantopoulos · Michael Roth · Andreas Symeonidis · Ewan Klein

Accepted: 19 January 2017

Abstract Mapping functional requirements first to specifications and then to code is one of the most challenging tasks in software development. Since requirements are commonly written in natural language, they can be prone to ambiguity, incompleteness and inconsistency. Structured semantic representations allow requirements to be translated to formal models, which can be used to detect problems at an early stage of the development process through validation. Storing and querying such models can also facilitate software reuse. Several approaches constrain the input format of requirements to produce specifications, however they usually require considerable human effort in order to adopt domain-specific heuristics and/or controlled languages.

We propose a mechanism that automates the mapping of requirements to formal representations using semantic role labeling. We describe the first publicly available dataset for this task, employ a hierarchical framework that allows requirements concepts to be annotated, and discuss how semantic role labeling can be adapted for parsing software requirements.

This work has been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

Themistoklis Diamantopoulos
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
E-mail: thdiaman@issel.ee.auth.gr

Michael Roth
School of Informatics, University of Edinburgh
Tel.: +44-131-6504665
Fax: +44-131-6506626
E-mail: mroth@inf.ed.ac.uk

Andreas Symeonidis
Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki
E-mail: asymeon@eng.auth.gr

Ewan Klein
School of Informatics, University of Edinburgh
E-mail: ewan@inf.ed.ac.uk

Keywords Semantic annotation · Software requirements · Semantic role labeling

1 Introduction

During the early stages of the software development lifecycle, developers and customers typically discuss and agree on the functionality of the system developed. This functionality is recorded as a set of functional requirements, which form the basis for implementations and the corresponding work plans, cost estimations and follow-up directives (van Lamsweerde, 2009).[1] Functional requirements can be expressed in various ways, including the use of UML diagrams and storyboards. However, most often they are expressed in natural language (Mich et al, 2004), as shown in example FR-1:

FR-1	The operator must be able to print the invoice.

Deriving formal specifications from functional requirements is one of the most important steps of the software development process. The final source code of a project depends on an initial model (e.g. described by a class diagram) that has to be designed very thoroughly in order to be functionally complete. However, designing such a model from scratch or even using requirements to do so is not a trivial task. While requirements expressed in natural language have the advantage of being intelligible to both clients and developers, they can also be ambiguous, incomplete and inconsistent. While several formal languages have been proposed to eliminate some of these problems, customers rarely possess the technical expertise for constructing and understanding highly formalized requirements. Thus, investing effort into automating the process of translating requirements to specifications can be highly cost-effective, as it removes the need for customers to learn new languages and/or for designing complex models from scratch.

To benefit from the advantages of both natural language and formal representations, we propose to induce the latter automatically from text as a semantic parsing task. For instance, given the software requirement in example FR-1, we would like to construct a representation that specifies the types of the entities involved (e.g., `Object`(invoice)) and captures explicit and inferable relationships among them (e.g., `owns`(operator, invoice)).

We argue that such formal representations are helpful in detecting ambiguities and errors at an early stage of the development process (e.g. via logical inference and verification tools), thus avoiding the costs of finding and fixing problems at a later, more expensive stage (Boehm and Basili, 2001). In addition, given a requirements document that describes an already implemented software component, formal representations can be employed to store the concepts and relations (along with their corresponding implementation)

[1] A system also involves *non-functional* requirements that describe quality criteria. However, this paper focuses on functional requirements, which specify what a system can do.

in a software repository. Such a repository could be employed, for example, to retrieve and reuse existing components in projects with similar requirements.

In the field of requirements modeling, there are several techniques for mapping requirements to specifications, however they mostly depend on domain-specific heuristics and/or controlled languages. Conversely, research in semantic parsing has applied state-of-the-art methods to various domains, but has so far ignored the challenges of analysing software requirements text. In this paper, we propose an automated way of semantically annotating functional requirements, i.e. identifying them as concepts in an ontology designed to produce system specifications. We define and implement a semantic parsing module that recognizes the main concepts of functional requirements (e.g. actors, objects, etc.) on the basis of a set of syntactic and semantic features. The module is trained from functional requirements instances, which are annotated with labels from a hierarchical conceptual framework.

The remainder of this article is structured as follows. In Section 2, we outline the wider project that provides motivation for our work. In Section 3, we describe previous work on mapping software requirements to formal representations. We provide a description of our concept ontology in Section 4. The tools that we developed to automatically process requirements and revise annotations are described in Sections 5 and 6, respectively. Finally, our methodology is evaluated using an annotated dataset of software requirements in Section 7 and we conclude in Section 8 with a discussion and outlook on future work.

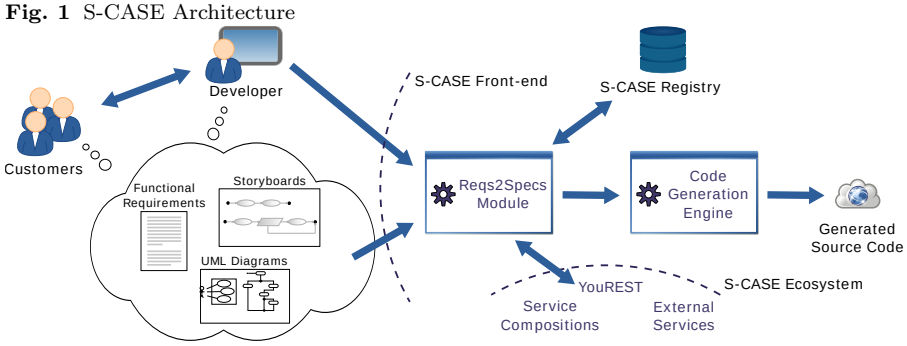
2 The Big Picture: the S-CASE Framework

The work described in this article is part of the EU-funded S-CASE (**S**caffolding S**ca**lable **S**oftware S**e**rvices) project.[2] S-CASE aims to provide a cloud-based suite of services and tools to support rapid software prototyping based on user requirements and system models provided in multimodal formats (Zolotas et al, 2016). The tools will equip developers with a set of artefacts that implement various aspects of the application to be developed. The S-CASE paradigm aspires to introduce new business models for service providers and Small and Medium-sized Enterprises (SMEs) who wish to provide integrated solutions while minimizing development cost and effort. The S-CASE conceptual architecture is shown in Figure 1 and comprises the following main parts:

- The **S-CASE Front-end** offers developers a set of tools to input multimodal representations requirements of the software application, reflecting the needs of their customers.
- The **Reqs2Specs Module** receives requirements expressed in various modalities and translates them into system specifications.
- The **S-CASE Registry** is a cloud-based storage scheme of reusable software artefacts (such as role, business logic and module entities), together with existing project templates, semantically annotated and interconnected.

[2] `http://www.scasefp7.eu`

Fig. 1 S-CASE Architecture



– The **Code Generation Engine** employs Model-Driven Engineering technology to produce source code from specifications using a series of transformations.
– The **S-CASE Ecosystem** is a cloud platform that provides all the available assets as services. It allows searching for external services and/or designing service compositions in order to fulfill the functionality not already covered by the code generation engine. Additionally, through the **YouRest** tool, it enables new proprietary/open source service and software asset providers to register their services/assets in the S-CASE Ecosystem.

The outcome of S-CASE are functional RESTful services. In order to address the problem that most software projects fail due to incomplete or misleading requirements, S-CASE supports the automated transformation of requirements into semantically annotated software artefacts.

The focus of this paper is the S-CASE **Front-end** and **Reqs2Specs** components, which control the user input in building a software system. In this context, we have designed and instantiated an ontology that captures the static view (requirements, use case models) of software projects (Section 4); designed and implemented a module that automatically translates requirements written in natural language text into a formal representation (Section 5), and developed a front-end that allows developers to enter and edit requirements in an easy manner (Section 6).

3 Related Work

A range of methods have been proposed to (semi-)automatically process requirements written in natural language text and map them to formal representations. Abbott (1983) was the first to introduce a technique for extracting data types, variables and operators from informal text. The proposed method follows a simple rule-based setup, in which common nouns are identified as data types, proper nouns as objects and verbs as operators between them. Booch (1986) described a method that extends Abbott's approach to object-oriented development. Saeki et al (1989) implemented a prototype that automatically

constructs object-oriented models from informal requirements, based on automatically extracting nouns and verbs. Although the authors found resulting object diagrams of reasonable quality, they concluded that human intervention was still necessary to distinguish between words that are relevant for the model and irrelevant nouns and verbs. Nanduri and Rugaber (1995) proposed to further automate object-oriented analysis of requirements by applying a syntactic parser and a set of post-processing rules. In their work, an analysis is made between automatically induced and hand-written object diagrams, indicating a need for more robust syntactic processing and human intervention to resolve ambiguities. In a similar setting, Mich (1996) employed a full NLP pipeline that contains a knowledge-base driven semantic analysis module, thus omitting the need for ad-hoc post-processing.

More recent approaches include those by Harmain and Gaizauskas (2003) and Kof (2004), who rely on a combination of NLP components and human interaction. Whereas most approaches aim to derive object diagrams, Gervasi and Zowghi (2005) map natural language requirements to logical forms in order to identify inconsistencies. Their method is based on a combination of morphosyntactic analysis and a shallow domain-specific parser, whose output is treated as a first-order predicate logic representation. A similarly logic-oriented approach by Ghosh et al (2014) employs broad coverage syntactic parser and a set of heuristic post-processing rules to induce representations in first-order temporal logic.

A considerable amount of research in requirements engineering nowadays is still dedicated to controlled languages, authoring rules and pattern-based specifications (Denger et al, 2003; Konrad, 2005; Tjong et al, 2006). Accordingly, some approaches to requirements processing define domain-dependent grammars and rules to specify how predefined constructions can directly be mapped to formal representations. For example, Post and Hoenicke (2012) describe a tool where requirements can be written according to a set of patterns, which are then automatically compiled into logic formulas and checked for consistency errors. Going beyond consistency checks and object diagrams, Gordon and Harel (2009) define a controlled fragment of English and a grammar that can map requirements expressed in this fragment into a formal language for specifying dynamic system behavior.

While there exists little previous work on analyzing software requirements using semantic parsing, various methods have been proposed for related tasks in NLP. 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 labor-intensive and scales poorly. 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 labeling (SRL)

techniques that assign relations (i.e., who did what to whom) to words-spans based on syntactic structure (Gildea and Jurafsky, 2002).

Extracting semantic relations between entities is one of the most well-known problems of NLP (Bach and Badaskar, 2007; Bunescu and Mooney, 2005b). Current approaches include feature-based and kernel-based methods. Given a dataset with annotated relations, feature-based methods involve extracting syntactic and semantic features from text, and providing the feature set to a classifier which is trained in order to identify relations. Several features have been employed, including e.g., the type of each entity, the number of words between the entities, etc., as well as several classifiers, including e.g. log-linear models (Kambhatla, 2004) or SVMs (Zhao and Grishman, 2005; GuoDong et al, 2005). However, feature-based methods are bound by heuristically choosing the features for the task at hand and kernel-based methods have been developed to overcome this limitation. Instead of manually extracting the features, these methods map text entities and relations in a higher dimensional space and classify relations according to their new representation. Typical representations include bag-of-words kernels (Bunescu and Mooney, 2005a) and tree kernels (Zelenko et al, 2003; Culotta and Sorensen, 2004; Bunescu and Mooney, 2005b), which usually result in features relevant to the context of the relations (e.g., words before, in the middle, or after a relation). Not surprisingly, tree kernels have proven to be more effective, since they include structural information (Bach and Badaskar, 2007; Culotta and Sorensen, 2004), and they are also quite efficient (Bunescu and Mooney, 2005b).

As discussed in this section, there are currently several approaches that focus on modeling functional requirements. However, these approaches are usually restricted to formalized languages or depend on heuristics and domain-specific information. By contrast, we propose to employ semantic role labeling techniques in order to learn to map functional requirements concepts to formal specifications. Although these types of techniques have been used extensively, their effectiveness on the problem domain of textual requirements has not been assessed. And, while kernel-based methods have been known to perform adequately on certain domains, current literature indicates that designing a method specifically for the domain at hand is optimal (Bach and Badaskar, 2007).

Consequently we have designed and implemented a semantic role labeler that employs a feature-based relation extraction method. Upon defining a set of features that describe the concepts found in each functional requirement, we employ an annotation scheme in order to produce annotated instances that are subsequently used to train a semantic role labeler.

4 Adding Semantics to Software Requirements

This section presents the design of an ontology for storing information derived from functional requirements. Ontologies provide a structured means of orga-

nizing information, underpin methods for retrieving stored data via queries, and allow reasoning over implied relationships between data items.

The Resource Description Framework (RDF)[3] provides one such formal framework of representing information. The RDF data model has three object types: *resources*, *properties*, and *statements*. A resource is any "thing" that can be described by the language, while properties are binary relations. An RDF statement is a triple consisting of a resource (the subject), a property, and either a resource or a string as object of the property.

Since the RDF data model defines no syntax for the language, RDF models can be expressed in different formats; the two most prevalent are XML and Turtle.[4] RDF models are often accompanied by an RDF schema (RDFS)[5] that defines a domain-specific vocabulary of resources and properties. Although RDF is an adequate basis for representing and storing information, reasoning over the information and defining an explicit semantics for it is hard. RDFS provides some reasoning capabilities, but is intentionally inexpressive, and fails to support a logical notion of negation. The Web Ontology Language (OWL)[6] was designed as a more expressive formalism which allows classes to be defined axiomatically and supports consistency reasoning over classes. OWL is built on top of RDF, but includes a richer syntax with features such as cardinality or inverse relations between properties. In the context of our work we employ OWL for representing information, since it is a widely-used standard within research and industry communities.[7]

4.1 Ontology Overview

Our work focuses on the static aspects of requirements elicitation, i.e., functional requirements and use case diagrams. The ontology design revolves around the concept of an agent performing some action(s) on some object(s). This simple binary structure is straightforward to express in RDF, and corresponds to Subject-Verb-Object sentences in natural language as well as UML use case diagrams.

4.1.1 Ontology Class Hierarchy

The class hierarchy of the ontology is shown in Figure 2. Anything entered in the ontology (any `owl:Thing`) is a `Concept` (since all ontology terms have the same namespace URI, we omit it for brevity). Instances of class `Concept` are further classified into `Project`, `Requirement`, `ThingType`, and `OperationType`.

[3] `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`

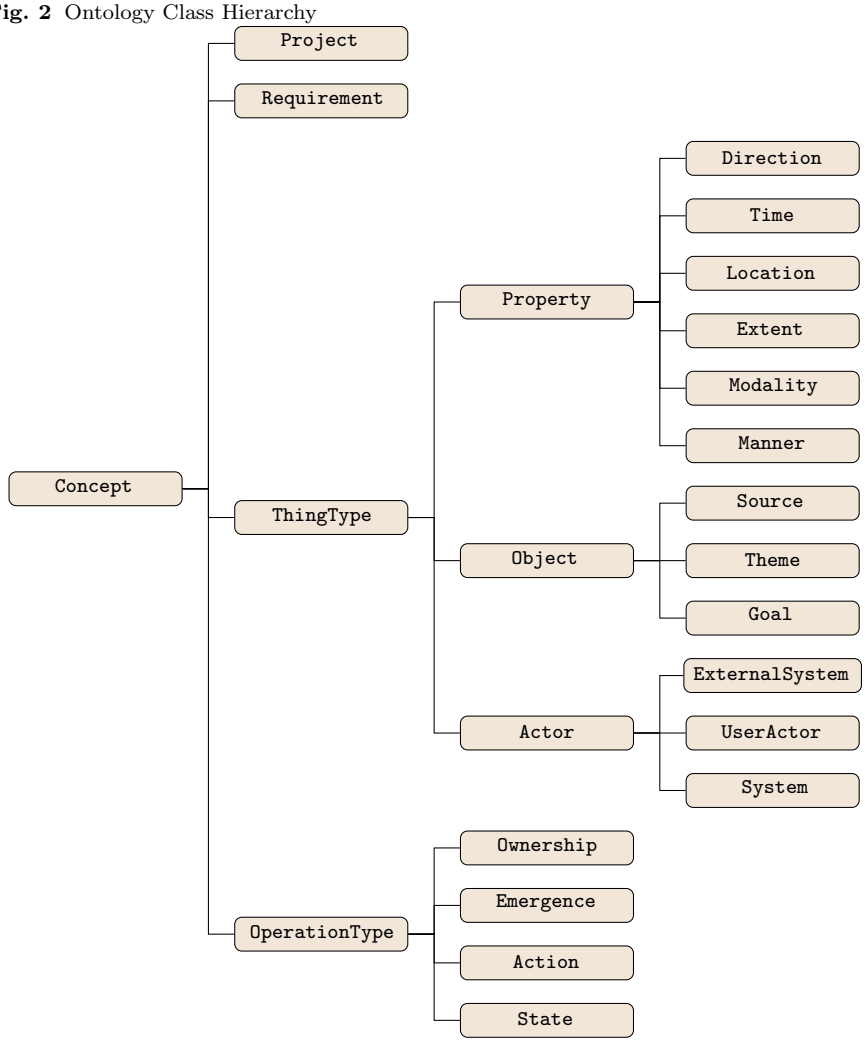
[4] `http://www.w3.org/TR/turtle/`

[5] `http://www.w3.org/TR/1999/PR-rdf-schema-19990303/`

[6] `http://www.w3.org/TR/2004/REC-owl-guide-20040210/`

[7] Although we don't rely on OWL inference capabilities explicitly in this paper, they are useful for expressing integrity conditions over the ontology, such as ensuring that certain properties have inverse properties (e.g. `owns`/`owned_by`).

Fig. 2 Ontology Class Hierarchy



`Project` refers to the software project under consideration, while `Requirement` represents the requirements of the system. Since each project has several requirements and each requirement introduces several other concepts, one may reconstruct the main structure of each project including its requirements using these concepts.

`ThingType` and `OperationType` are the main types of entity found in any requirement. The former refers to entities which are actors or are acted upon, while instances of `OperationType` are actions performed by actors on other entities. In detail, a `ThingType` instance can be one of the following classes:

- `Actor` refers to the actors of the project. It has the subclasses `UserActor`, `ExternalSystem` and `System`, referring to the users, the external systems interacting with the system, and the system itself respectively.
- `Object` involves any object or resource of the system that is acted upon. It has the subclass `Theme` that covers most instances, and the subclasses `Source` and `Goal` that involve `Object` instances that relate to some other `Object`. For example, in the phrase "get tag from bookmark", "bookmark" is a `Source`, whereas in the phrase "assign tag to bookmark", it is a `Goal`.
- `Property` includes all modifiers of objects/actions that assign some property to the object/action involved. Taking inspiration from the PropBank project (Palmer et al, 2005), we define several subclasses to disambiguate among the modifier types with different semantics. These are `Direction` (e.g. "navigate *North*"), `Time` (e.g. "show a message *when the system is busy*"), `Location` (e.g. "have an exit button *at the main menu* of the system"), `Extent` (e.g. "exit if no action is performed *for 30 minutes*"), `Modality` (e.g. "a bookmark *that can be deleted*"), and `Manner` (e.g. "search a bookmark *by tag*").

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 usually expressed using the verb "have", e.g. "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, e.g. "the bookmark is *re-indexed*".
- `Action` describes an operation performed by an `Actor` on some `Object`, e.g. "The user must be able to *create* a bookmark".
- `State` indicates an operation that describes the status of an `Actor`, e.g. "the user is *logged in*".

Finally, note that although the ontology was designed from scratch, it can be easily connected to existing ontologies. Several of our ontology terms have counterparts in other well-known vocabularies; for example, `Project` corresponds to `doap:Project`.[8] To simplify presentation, the ontology is presented here without reuse of existing vocabulary, though we aim to review the utility of adopting an approach more compatible with Linked Data principles in future.

4.1.2 Ontology Properties

The relations between the ontology (sub)classes are very important, since they define the possible interactions between the different concepts. We define a set of properties in order to adequately cover all possible interactions. To begin with, we define the high-level properties of requirements as shown in Table 1. We use the term 'high-level' for the properties shown in Table 1

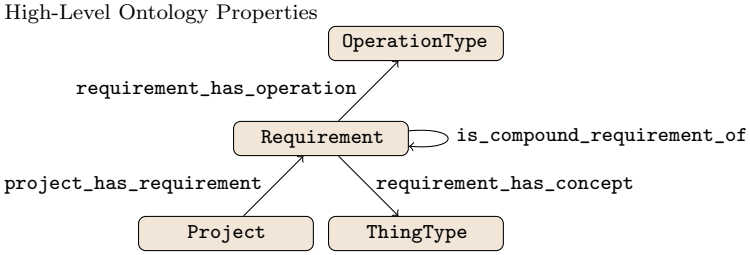
[8] `https://github.com/edumbill/doap/wiki`

Table 1 High-level properties of the ontology

OWL Class	OWL 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`
`Requirement`	`requirement_has_concept`	`ThingType`
`ThingType`	`is_concept_of_requirement`	`Requirement`
`Requirement`	`requirement_has_operation`	`OperationType`
`OperationType`	`is_operation_of_requirement`	`Requirement`

since they cover the interactions among the four main classes of the ontology (`Project`, `Requirement`, `ThingType`, `OperationType`). Each project can have many different requirements while each requirement can also be compound, i.e. contain other requirements. In addition, each requirement comprises several `ThingType` and `OperationType` instances. Furthermore, since RDFS allows us to define subproperties, `requirement_consists_of` can be further refined in subproperties `requirement_has_concept` and `requirement_has_operation`, for `ThingType` and `OperationType` instances respectively. Similarly, the property `consist_requirement` has the subproperties `is_concept_of_requirement` and `is_operation_of_requirement`, respectively. The defined properties are visualized in Figure 3, including only one of the two directions for simplicity.

Fig. 3 High-Level Ontology Properties



Apart from high-level properties, we also define 'low-level' properties as the properties that cover the interactions among the different subclasses of `ThingType` and `OperationType`. These properties are depicted in Table 2.

The structure of these properties is similar to the way sentences are structured. Specifically, instances of type `actor` are actors of operations, i.e. they are connected with `OperationType` instances via `is_actor_of` and `has_actor`. Operations that are transitive connect also to objects. 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`

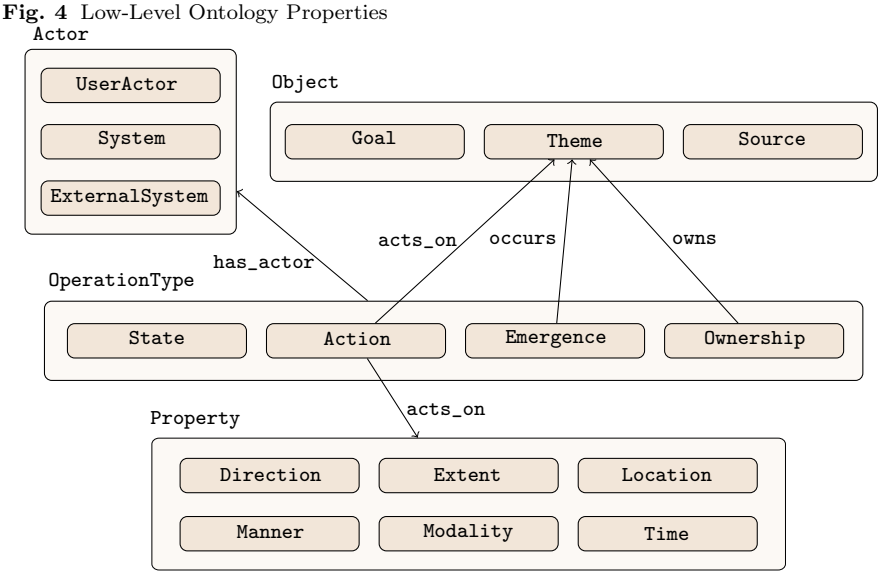
Table 2 Low-level properties of the ontology

OWL Class	OWL 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

properties. The non-transitive State operation connects only with an Actor (via is_actor_of and has_actor).

Finally, the composite nature of objects is also described using properties. An Object can have a Source and/or a Goal. It connects with the former via has_source and is_source_of, and with the latter via has_goal and is_goal_of. These subclasses and their properties are shown in Figure 4.

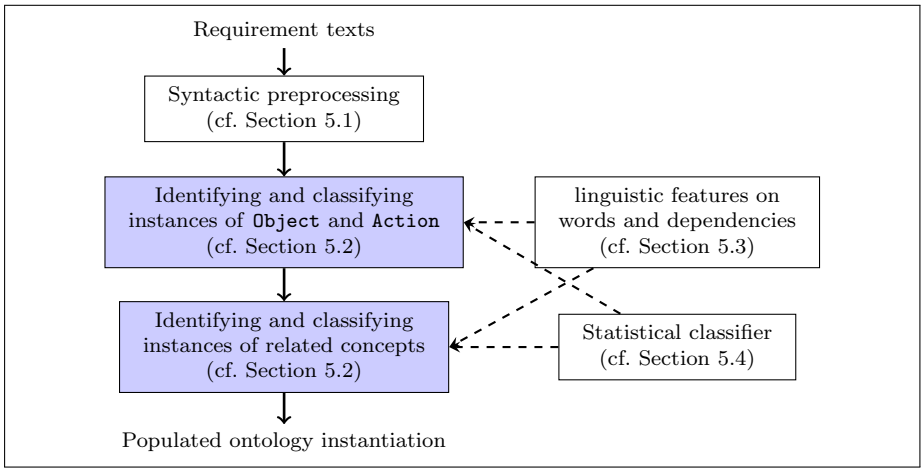
Fig. 4 Low-Level Ontology Properties



5 A Parser for Transforming Software Requirements

Based on the modeling effort described in the previous section, we have developed a parser that learns to automatically map software requirements written in natural language texts to concepts and relations defined in the ontology.[9] We identify instances of the concepts `Actor`, `Action`, `Object`, and `Property`, and the relations among them (`is_actor_of`, `acts_on`, and `has_property`). These concepts were selected since they align well with key syntactic elements of the sentences, and are intuitive enough to make it possible for non-linguists to annotate them in texts. (See subsection 6.2 for more details on our choice of these concepts). Our parser, however, could also identify more fine-grained concepts, given more detailed annotations. In practice, the parsing task involves several steps: first, concepts need to be identified and then mapped to the correct class and, second, relations between concepts need to be identified and labeled accordingly. We employ a feature-based relation extraction method, since we are focussing on a specific problem domain and type of input, i.e. textual requirements of software projects. This allows us to implement a parsing pipeline based on previous work in semantic role labeling (cf. Figure 5).

Fig. 5 Pipeline architecture for parsing software requirements



Our approach was motivated by initial analysis of a small set of software requirements where we tested several previous methods and found a combination of dependency parsing and semantic role labeling techniques to generally provide the best off-the-shelf results. Other approaches turned out to generalize worse or did not provide robust output. As an example, applying

[9] We first introduced this parser in (Roth and Klein, 2015). This section provides additional details of the parsing architecture and underlying motivations.

the categorial grammar based C&C parser (Clark and Curran, 2007) to our dataset led to higher error rates and coverage gaps due to differences in domain and a high ratio of out-of-vocabulary words. The advantage of using a broad-coverage syntactic dependency parser is that it produces robust output, which can be used to train a separate semantic role labeling model. To model the constraints and characteristics of the software requirements domain, we started from techniques developed for labeling semantic roles that are motivated on generic linguistic grounds and adapted them to use the concepts and relations defined in the ontology (cf. Section 4).

The following subsections describe our implementation in more detail. In Section 5.1, we introduce the preprocessing pipeline that allows us to compute a syntactic analysis for each English sentence in our corpus that expresses a requirement. Section 5.2 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. We define the features and learning techniques applied to train each statistical model in subsections 5.3 and 5.4, respectively. We illustrate our analysis using the following two examples:

FR-2	The user must be able to upload photos.

FR-3	Any user must be able to search by tag the public bookmarks of all RESTMARKS users.

5.1 Syntactic Analysis

The syntactic analysis stage of our pipeline architecture performs the following steps: 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).

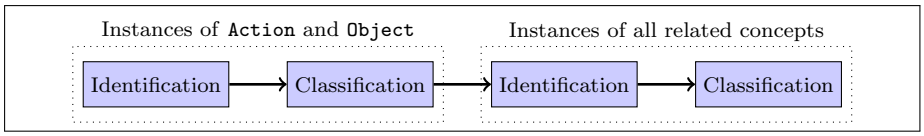
For all syntactic analysis steps, we rely on components and pre-trained models from a system called Mate Tools (Björkelund et al, 2009; Bohnet, 2010), which is freely available online.[10] This choice is based on three criteria: (1) the system achieves state-of-the-art performance on a benchmark dataset for syntactic analysis (Hajič et al, 2009), (2) the output of the syntactic analysis has been successfully used as input for the related task of PropBank/NomBank-style semantic role labeling (Palmer et al, 2005; Meyers et al, 2008), and (3) the system is fast and robust, meaning that it can be integrated efficiently into the overall software framework of S-CASE (cf. Section 2).

[10] http://code.google.com/p/mate-tools/

5.2 Semantic Analysis

The semantic analysis subsystem adopts a pipeline architecture for taking the output of syntactic preprocessing and extracting instances of ontology concepts and relations, as shown in Figure 6.

Fig. 6 Components of our SRL-inspired semantic analysis module



The semantic analyser comprises two main modules, one for `Action` and `Object` instances, and the other for all related concepts (indicated with dotted lines in Figure 6). Each module is further broken down into identification and classification components.Semantic analysis is carried out in four steps: (1) identifying instances of `Action` and `Object`; (2) allocating these to the correct concept (either `Action` or `Object`); (3) identifying instances of related concepts (i.e., `Actor` and `Property`) and (4) determining their relationships to concept instances identified in step (1). Our method is based on the semantic role labeler from Mate Tools and uses the built-in re-ranker to find the best joint output of steps (3) and (4). We extend Mate Tools with respect to continuous features and arbitrary label types. We describe each component of our implementation in the following paragraphs.

Step (1) The first step of the pipeline identifies words in a text that are recognized as either of the ontology concepts `Action` and `Object`. The motivation for identifying these two concept types first is that only they govern relationships to all other ontology concepts through the three relations `acts_on`, `has_actor` and `has_property`. We hence expect the corresponding linguistic units to behave similarly to PropBank/NomBank predicates and can apply similar features as used in the *predicate identification* step implemented in Mate Tools (Björkelund et al, 2009). Our implementation considers each verb and each noun in a sentence and performs binary classification (see subsection 5.4) to determine whether these instances are predicates based on lexical semantic and syntactic properties.

Step (2) The next step determines which ontology concept is applicable to each instance identified in Step (1). That is, for each verb and noun in a sentence classified as a potential instance of `Action` or `Object`, the component predicts and instantiates the actual ontology concept (e.g., "upload"→`Action`, "search"→`Action`). As in the previous component, lexical semantic and syntactic properties are exploited to perform classification. This step corresponds

to the *predicate disambiguation* step applied in PropBank/NomBank semantic role labeling, with the only difference being that we always map to the same ontology concepts and not to fine-grained concepts that are defined for each predicate.

Step (3) The component for determining related concept instances detects words and phrases in a text that are related to the instances previously identified in Step (1). The main goal of this step is to identify the `Actor` of an `Action` and affected `Object`s as well as instances of `Property` that are related to any of the former. As such, this step is similar to *argument identification* in semantic role labeling. Accordingly, we take as input potential 'arguments' of a concept instance from Step (1) and perform binary decisions that indicate whether a word or phrase instantiates a (related) ontology concept. In example FR-2, both "the user" and "photos" are ontology instances that are related to the `Action` expressed by the word "upload". In example FR-3, instances related to "search" are: "any user", "by tag" and "the public bookmarks of all RESTMARKS users". In this example, "of all RESTMARKS users" is further related to the `Object` expressed by the phrase "the public bookmarks".

Step (4) Finally, the component for labeling relationships determines which relations hold between a pair of instances as identified in Steps (1) and (3). Generally, each instance can be involved in multiple relations and hence more than one concept type can apply to a single entity. To represent this appropriately, the component performs classification on pairs of related instances (e.g., ⟨"the user", "upload"⟩ → ⟨`Actor`, `Action`⟩, ⟨"by tag", "search"⟩ → ⟨`Property`, `Action`⟩). This step roughly corresponds to the *argument classification* step of the semantic role labeler implemented in Mate Tools. As with concept labels, however, our set of potential relations is predefined in the ontology.

For these identification and classification steps, our implementation relies on lexical semantic and syntactic properties as well as additional characteristics that hold between the linguistic expressions that refer to the considered instances (e.g., their order in text). More details regarding the features applied in each step are described in the next subsection.

5.3 Features

In practice, each step in our pipeline is implemented as a logistic regression model that uses linguistic properties as features, for which appropriate features weights are learned based on annotated training data. The majority of features applied in our models are already implemented in Mate Tools (Björkelund et al, 2009). Given that the number of annotations available for our task is about one order of magnitude smaller than those in PropBank/NomBank, we utilize a subset of features from previous work, as summarized in Table 3, which we greedily selected based on classification performance.

Table 3 Linguistic properties that are used as features in statistical classification

	`Action` and `Object`		Related concepts	
	identification	classification	identification	classification
Affected word forms	•	•	•	•
Affected word lemmata	•	–	–	–
Word part-of-speech	•	–	•	•
Word vector representation	•	•	•	•
Relation to parent	•	–	•	•
Parent part-of-speech	•	•	–	–
Set of dependent relations	–	•	–	–
Single child words	•	–	–	–
Single child part-of-speech	•	–	–	–
Dependencies between words	–	–	•	•
Order of affected words	–	–	•	•
Distance between words	–	–	•	–

As shown in Table 3, different features have been proven optimal for each classification task. The ones selected here are compatible with the current literature on semantic relation extraction (Gildea and Jurafsky, 2002; Pradhan et al, 2004). For instance, the part-of-speech of the word proved to be useful for identifying all instances and classifying related concepts of `Action` or `Object`, whereas the part-of-speech of its parent turned out to be useful only for identifying and classifying instances of `Action` or `Object`. Also, several features are only defined for related concepts as they are actually dependent on relations, e.g., the position of the related concept, i.e. before or after the `Action` or `Object`.

To compensate for sparse features in our setting, we define additional features based on distributional semantics. The motivation for such features lies in the fact that indicator features and feature combinations (e.g. the affected word type plus its part-of-speech) can be too specific to provide robust generalization for semantic analysis. To overcome the resulting gap in coverage, we represent each word in a classification decision by a low-rank vector representation that is computed based on word-context co-occurrence counts and can be computed over large amounts of unlabeled text. As distributional representations tends to be similar for words that are similar in meaning, this allows word type information to be utilized at test time, even if a specific word has not occurred in the training data.

As indicated in Table 3, we apply vector representations of words for identifying instances of `Action` and `Object` as well as for classifying instances of related concepts. Following a recent comparison of different word representations for semantic role labeling (Roth and Woodsend, 2014), we use a set of publicly available vectors that were learned using a neural language model (Bengio et al, 2003).[11]

[11] `http://github.com/turian/neural-language-model`

5.4 Learning

For each component of our parser, we rely on the annotated training data described in Section 7 to learn a logistic regression classifier, using the LIBLINEAR toolkit (Fan et al, 2008). The underlying statistical computation in this toolkit is performed by iteratively optimizing the feature weights in the form of a weight vector $\mathbf{w}$ given feature values as a vector $\mathbf{x}$ and a single binary classification label y following equation (1):

$$\min_w \log(1 + e^{-y\mathbf{w}^T\mathbf{x}}) + \frac{1}{2}\mathbf{w}^T\mathbf{w} \quad (1)$$

The first part of equation (1) is the logistic loss, which is used to minimize the feature weights $\mathbf{w}$, such that the output of the logistic function applied to $\mathbf{w}^T\mathbf{x}$ is close to 1 iff y=1 ($\mathbf{w}^T$ denotes the transpose of vector $\mathbf{w}$). 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. We apply our preprocessing components described in Section 5.1 to extract feature values and to derive class labels from annotated instances of concepts and relations. In the identification steps, we use the class label 1 to indicate that a word expresses an instance of an ontology concept (otherwise it is -1). In the case of multi-way classification decisions, a one-vs-all model is learned for each concept in the ontology.

6 Semi-automated Annotation

In this section, we provide additional details about the annotation scheme, as applied to the dataset introduced in Section 7, and describe the annotation tool that we designed and implemented for the task of annotating functional requirements.

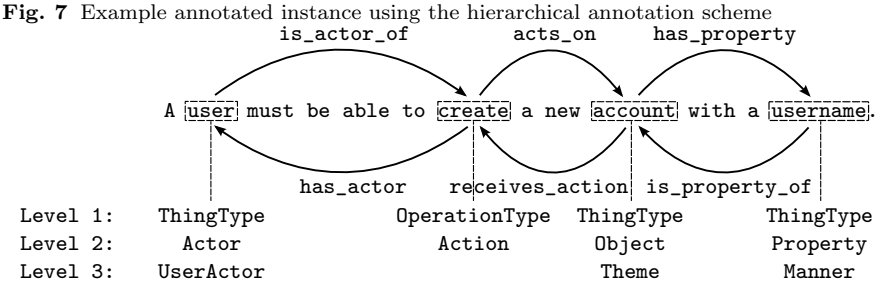
6.1 Annotation Scheme

A primary question in annotating requirements is deciding how complex the annotations should be. Ideally, an annotation scheme would be as close as possible to the ontology classes described in Section 4, since that represents our final desired result. However, such a scheme would be very difficult for annotators to implement without considerable background knowledge.

As a result, we have adopted a multi-step annotation scheme (Roth et al, 2014) in which decisions made at one iteration are further refined in later iterations. Given the class hierarchy introduced in Section 4, it is natural to split up the annotation iterations so that each one corresponds to a level in the ontology. Note that "level" here refers to the depth of each concept in the ontology (not to be confused with the level of the properties defined in Section 4). This means that in the first iteration, we ask annotators to simply

mark all instances of `ThingType` and `OperationType`, while in the second iteration they are asked to mark instances of `Actor`, `Object`, `Action`, 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. the phrase "a user can delete his/her account" involves not only an `Action` performed on "account" but also `Ownership` of "account" by "user"). Consider the example of Figure 7.

Fig. 7 Example annotated instance using the hierarchical annotation scheme



In this sentence, the first iteration would include annotating "user", "account" and "username" as instances of `ThingType` and "create" as an instance of `OperationType`. The second iteration would include annotating "user" as an `Actor`, "create" as an `Action`, "account" as an `Object`, and "username" as a `Property`. The next iteration after that would involve specifying "user" as a `UserActor`, "account" as a `Theme`, and "username" as a `Manner`. Finally, in this example we could also add one more iteration where we would specify "account" as an object `owned_by` "user". While this relation is not explicitly given in this sentence, it is in fact correct.

6.2 Annotation Process

The task of annotation was performed by two annotators using a tool designed for this purpose (see subsection 6.3). The annotation process was designed to optimise the trade-off between the amount of information collected and the level of agreement between the annotators—more complex annotation schemes tend to increase disagreement. For this reason, we focussed on annotation at the second level of the hierarchical scheme described in the previous subsection. The concepts to be identified are `Actor`, `Action`, `Object`, and `Property` and the relevant relations `is_actor_of`, `acts_on`, and `has_property`. These four concepts effectively cover the syntactic elements of the sentences and allow for intuitive annotation of software requirements. Note that we initially ran a round of annotations using all levels in the ontology, but the results

demonstrated an unacceptably high rate of disagreement between different annotators. This is unsurprising as certain concepts are hard to distinguish even for experts. For example, difficulties arose for subconcepts of `Property`: distinguishing between cases of `Manner` and `Modality` or between `Time` and `Extent` provoked discussion not only between the annotators but also between the authors of this work.

In view of these considerations, the annotators were instructed to annotate instances of `Actor`, `Action`, `Object`, and `Property`, as well as the relevant relations among them. They were given the following guidelines:

- Each word that denotes an action should be annotated as an instance of the `Action` class. If a verb is complemented by a noun phrase that provides further specification, the noun phrase should be annotated as well (e.g., the phrase "perform search" will be annotated as a whole).
- All words that denote concepts related to an action should be identified and, depending on their relation to the respective action, annotated using a suitable class (`Actor`, `Object` or `Property`) and linked via a corresponding relation (`is_actor_of`, `acts_on` or `has_property`, respectively).
- If a concept is further specified by other words or word spans in a given requirement, these should be annotated as instances of `Property` and linked to the respective concept instance.
- If it is unclear whether a concept denotes a `Property` of an `Action` or an `Object` affected by an `Action`, the decision will be based on syntactic information (e.g. prepositional attachment→`Property`, indirect object→`Object`).
- If two concepts denote related instances of `Property` and the direction of the relation is unclear, the ambiguity will be resolved based on the order of the respective words in the syntactic dependency tree.

6.3 Annotation Tool

As mentioned earlier, our parsing approach relies on the availability of a sufficient number of annotated examples for training purposes. Since annotating can be a hard task, especially for inexperienced users, we created a bespoke *S-CASE Requirements Annotation Tool* to ease the process. It focusses on the second level of the annotation scheme defined in the previous subsection, i.e. asking users to define actors, actions, objects and properties.

In the context of the S-CASE project, the tool can be used by a developer to retrain the parser so that it is adapted to his/her (possibly specialized) requirements. Hence, it serves as an external tool that can be used to retrain the parser. However, it can also be used to provide a clear view of the way requirements are parsed, and allow the developer to further improve the annotations. A running version of the tool is available at:

`http://rat.scasefp7.com/`

6.3.1 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. The tool allows the user to specify terms (or phrases) as one of the entities Actor, Action, Object, or Property. For relations between these terms, we define three relations: IsActorOf, ActsOn, and HasProperty. IsActorOf is declared from Actor to Action, and ActsOn is defined from Action to Object or from Action to Property. Finally, HasProperty is declared from Actor to Property, or from Object to Property, or from Property to Property.

Mapping the annotated terms to the ontology is quite straightforward. Entity annotations Actor, Action, Object, and Property are mapped to the `Actor`, `Action`, `Object`, and `Property` ontology classes, respectively. Relation annotations, i.e., IsActorOf, ActsOn, and HasProperty are mapped to the corresponding ontology properties, instantiating also the inverse properties, e.g., for IsActorOf both `is_actor_of` and `has_actor` are instantiated. Also, since the identifiers of the requirements and the project are known, `Project` and `Requirement` are also instantiated, including their related properties. Keeping the annotations to a minimum, we denote each file as an annotated `Project` instance, while each sentence corresponds to an instance of `Requirement`.

Notice that we refrain from also declaring the inverse relations (e.g. HasActor) in order to keep the tool as simple as possible. Consequently the tool presents a very simple task to the user; there are only four entities and three relations, with all of them are quite close to their informal meanings in English. More specifically, the triple Actor-Action-Object is similar to Subject-Verb-Object (SVO), while Property mostly corresponds to modifiers (adjectives, prepositional phrases, etc.).

The tool also allows requirements to be annotated by our parser (cf. Section 5). This feature, which proved to be popular with annotators, means that the manual component of the task can be reduced to checking and correcting the machine-produced annotations. This process of semi-automatic annotation is faster and more efficient than carrying out the whole process by hand.

6.3.2 Example Annotated Project using the Annotation Tool

In this subsection, we provide an example of using the annotation tool for the requirements of project *Restmarks*. Restmarks is a demo project created in the context of S-CASE to serve as a scenario of the S-CASE workflow. In particular, Restmarks can be seen as a social service for bookmarks. The users of the service can store and retrieve their bookmarks, share them with the community and search for bookmarks by using tags. A screenshot of the tool depicting the annotations for Restmarks is shown in Figure 8.

As shown in Figure 8, the annotations are comprehensive; even users with no experience should be able to identify and label the appropriate entities and relations. The tool exports annotations in different formats, including OWL and TTL. Table 4 illustrates the Restmarks entities discovered via the tool.

Fig. 8 Annotated requirements of project Restmarks

Table 4 Ontology instances for the entities of Restmarks

OWL Class	OWL Individual(s)
`Project`	`Restmarks`
`Requirement`	`FR1`, `FR2`, `FR3`, `FR4`, `FR5`, `FR6`, `FR7`, `FR8`, `FR9`, `FR10`, `FR11`, `FR12`, `FR13`
`Actor`	`user`, `users`
`Action`	`create`, `retrieve`, `mark`, `search`, `delete`, `add`, `providing`, `login`, `update`
`Object`	`account`, `bookmark`, `bookmarks`, `password_1`, `tags`
`Property`	`RESTMARKS`, `password`, `tag`, `public`, `private`, `logged_in_to_his_account`, `logged_in`, `username`, `user_1`

This example should further clarify the annotation scheme described subsection 6.1. Terms such as “bookmark” or “tag” are instances of `Object` and could be further refined as instances of `Theme` using ontology software such

as Protégé.[12] Similar refinement is possible from instances of `Actor` to the subclass `UserActor`, as well as from `Property` to its various subclasses. Operations are generally instances of `Action` since this is the most usual subclass of `OperationType`. Finally, the parser, as discussed in Section 5, can use synonym and type lexicons to find semantically-related terms such as "bookmark" and "bookmarks" and keep one of the two.

The tool also instantiates the properties of the ontology. For example, the properties for requirement FR4 of Restmarks are shown in Table 5. It can be observed that properties omitted from Table 5 include the `has_goal` and `is_goal_of` properties between the `bookmark` and `account` instances. However, as we mentioned, the tool focusses on the second level of the annotation scheme; the next levels can be handled using ontology software such as Protégé. Since relations are defined in the second hierarchical level, assigning different classes to certain instances is now simple.

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

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

As shown in this example, after using the tool to insert and annotate software requirements, the user is presented with an instantiation of the ontology that we described in Section 4. This instantiation is actually a model that corresponds to the specifications of the software project. As part of the software development process (see Figure 1), subsequent work may involve producing source code for the project or synthesizing already existing services that comply with these specifications. Consequently this model can be seen as an intermediate step between the requirements and the implementation of a software project. Given that it is comprehensive and highly intuitive, the user can either examine it manually or construct validation rules in order to ensure that the desired functionality is covered. For example, Table 4 provides a description of the Restmarks web service that includes the main specifications required to develop it. In particular, instances of concept `Object` correspond to the resources of the web service while instances of `Property` and `Action` are their parameters and allowed HTTP 'verbs', respectively. So, for an `account` endpoint, the user must be able to `create` it using the parameters `username` and `password`.

As well as supporting semi-automatic annotation of requirements and extraction of specifications, our method provides a traceable model. That is, given the `Requirement` and `Project` concepts that connect *via* properties to all instances of the ontology, the user can trace any concept instance back to

[12] `http://protege.stanford.edu`

the original requirement. As a result, any problems emerging at the phase of translating specifications to source code, e.g. omissions, duplicate or erroneous functionality, etc., can be detected early and traced back to the original requirements. For example, in Table 4, instances of `Object` include `password`, however the password of a user should not be a resource of the web service, both for functionality and for security reasons. In this case, the `password` instance can be traced back to the third requirement of Restmarks (see Figure 8), so the user can easily refine the annotations on this requirement to make `password` a parameter of `account` (i.e. by setting `account` as an Object, `password` as a Property and connecting the two instances using the HasProperty annotation).

7 Evaluation

As well as using annotated data to train our parsing system, we also need correctly annotated data as a gold standard against which to evaluate the performance of the system. Since there is no publicly available corpus of annotated textual requirements, we have published our own corpus at:

`http://issel.ee.auth.gr/software-algorithms/`

There is a fair amount of latitude in how requirements can be formulated in natural language, and consequently they can vary widely in quality, style and granularity. To cover a suitable range of variation, we asked lecturers from several universities to provide requirement documents written by students involving projects in various domains, such as embedded systems, virtual reality and web applications.[13] From these documents, we extracted lists of single sentence requirements. We also collected single sentence requirements from the S-CASE project (cf. Section 2), describing industrial prototypes of cloud-based web services. Since requirements documents typically follow standardised templates, it was straightforward to identify and collect the requirements sentences within them. Table 6 gives some overview statistics of the requirements we collected.

We observe that the number of requirements in student projects is much higher (270) than that of industrial prototypes (55). The token counts reveal however that requirements written for industrial prototypes are longer on average (16.6 vs. 11.6 words). This observation might be related to the fact that students in software engineering classes are often provided with explicit guidelines on how to concisely express requirements in natural language. As a consequence, we also find their requirement texts tend to be more regimented and stylized than those written by senior engineers. Requirements FR-4 and FR-5 show examples of student-written requirements, while FR-6 and FR-7 show examples of developer-written requirements.

[13] The majority of requirements collected in this way were provided by a software development course organized jointly by several European universities, cf. `http://www.fer.unizg.hr/rasip/dsd`

Table 6 Statistics on our requirements collection and existing semantic parsing datasets.

	#sentences	#tokens	#types
student projects	270	3130	604
industrial prototypes	55	927	286
Our dataset (total)	325	4,057	765
GeoQuery880	880	6,656	279
Free917	917	6,769	2,035

FR-4	The user must be able to vote on polls.

FR-5	The user must be able to evaluate an article.

FR-6	For each user contact, back-end must perform a check to determine whether the contact is a registered user or not.

FR-7	Back-end must perform dynamic updates to the friends list via silent notifications in case of social graph updates.

As already noted, no annotated textual requirements corpora are publicly available. Consequently, in Table 6, we provide a quantitative comparison of our corpus to the GeoQuery880 (Tang, 2003) and the Free917 (Cai and Yates, 2013) corpora. The GeoQuery880 corpus involves sentences used to query a geographical database, while Free917 comprises a set of logical queries used to query Freebase (Bollacker et al, 2008), a semantic graph database designed to store common knowledge. These two corpora were selected as they are extensively used by the current state of the art for semantic parsing tasks (Wong and Mooney, 2006; Berant et al, 2013). As indicated by the presented counts, our collection is still relatively small in terms of example sentences. However, the difference in the total number of tokens is not so large, given that sentences in our dataset are much longer on average. We further observe that the token/type ratio in our texts lies somewhere between the ratios reported in the work reported above. Based on the lexical variety and average sentence length that we observed, we expect our texts to be a challenging domain for existing semantic parsing approaches. In particular, the ratio between the number of lexical items and available sentences is likely to hinder the automatic learning of domain-specific grammars and lexicons(Zettlemoyer and Collins, 2007; Kwiatkowksi et al, 2010). It was for this reason that we resorted to the more traditional approach based on general purpose tools, as described in Section 5.

7.1 Analysis on Annotated Data

The annotation was performed by two annotators following the process defined in subsection 6.2. In total, the annotators marked 1,667 and 1,890 phrases

as ontology instances respectively. Following standard practice, we used the kappa coefficient (according to the Fleiss definition (Fleiss et al, 1981)) to measure inter-annotator agreement. In order compute this statistic, we reduce each annotation to the syntactic head word of a phrase and view each single word as an annotation instance. We also measure agreement for an additional *null* category that represents un-annotated words. The resulting per-category kappa scores are 0.82 for *null*, 0.96 for `Actor`, 0.91 for `Action`, 0.77 for `Object`, and 0.72 for `Property`. Raw agreement over all instances lies at 89%, with a chance-corrected agreement of 82%. As already noted, using more concepts resulted in higher disagreement between the annotators. For example, early annotations showed that the annotators were just as likely to disagree as agree for the subconcepts of `Property`. These subconcepts were therefore dropped from the task in order to ensure the reliability of the dataset.

For the concept categories that were kept, however, the numbers indicate that the two annotators substantially agreed on each category. The highest level of disagreement can be observed between the categories *null* and `Property` (196 instances) and between `Property` and `Object` (50 instances). The former type of disagreement mainly stemmed from the fact that both annotators applied different levels of granularity in their decisions. Given the phrase "web site", for example, one annotator marked the whole phrase as an ontology instance of type `Object`, whereas the other annotator only marked "site" as an `Object` and "web" as a `Property` thereof.

For the final dataset, we decided to always adopt the most fine-grained annotation by merging all annotations of concept instances. All disagreements involving two ontology classes were resolved in group discussions and manually added to our dataset. Table 7 provides statistics for all annotations after revision. Note that instances of `Actor` can occur with multiple different instances of `Action` and some instances of `Object` are not involved in any, hence the number of relations can differ from the number of associated concepts.

Table 7 Counts of annotated instances of concepts and relations after revision

Concept	Instances	Relations	Instances
`Action`	435		
`Actor`	305	`has_actor`	355
`Object`	613	`acts_on`	531
`Property`	698	`has_property`	690
Total	2,051	Total	1,576

7.2 Experiments

We evaluate the performance of the semantic role labeling approach described in subsection 5.2, using the annotated dataset described in Section 7. As evalu-

ation metrics, we apply labeled precision and recall. We define *labeled precision* as the fraction of predicted labels of concept and relation instances that are correct, and *labeled recall* as the fraction of annotated labels that are correctly predicted by the parser. To train and test the statistical models underlying the semantic analysis components of our pipeline, we perform evaluation in a 5-fold cross-validation setting. That is, given the 325 sentences from the annotated data set, we randomly create five folds of equal size (65 sentences) and use each fold once for testing while training on the remaining other folds.

As baselines, we apply two pattern-based models that are similar in spirit to earlier approaches to parsing software requirements (cf. Section 3). The first baseline simply uses word level patterns to identify instances of ontology concepts and relations. The second baseline is similar to the first but also takes into account syntactic relationships between potential instances of ontology concepts. For simplicity, we train both baseline models using the same architecture as our proposed method but use a subset of the applied features. In the first baseline, we only apply features indicating word forms, lemmata and parts-of-speech as well as the order between words. For the second baseline, we use all features from the first baseline plus indicator features on syntactic relationships between words that potentially instantiate ontology concepts. The results of both baselines and our full semantic role labeling model are summarized in Table 8.

Table 8 Performance of our full model and two simplified baselines; all numbers in %

Model	Precision	Recall	F_1-score
Baseline 1 (word-level patterns)	62.8	35.2	45.1
Baseline 2 (syntax-based patterns)	78.3	62.1	69.3
Full SRL model	77.9	74.5	76.2

Using all features described in Section 5.3, our model achieves a precision and recall of 77.9% and 74.5%, respectively. The corresponding F_1-score, calculated as the harmonic mean between precision and recall, is 76.2%. The baselines only achieve F_1-scores of 45.1% and 69.3%, respectively. A significance test based on random approximate shuffling (Yeh, 2000) confirmed that the differences in results between our model and each baseline is statistically significant ($p<0.01$).

As these metrics illustrate, our semantic parsing module is quite effective in terms of software engineering automation. In particular, the recall value indicates that our module correctly identifies roughly 75% of all annotated instances and relations. Additionally, the precision of our module indicates that approximately 4 out of 5 annotations are correct. Thus, we expect our parser to significantly reduce the effort (and time) required to identify the concepts that are present in functional requirements and annotate them accordingly. It was not possible within the timeframe of the project to objectively measure

whether tool support based on semantic parsing improved the overall productivity of software developers using S-CASE, and consequently this task will have to be the subject of future research. However, the informal feedback that we received about the utility of our parser from developers who participated in our pilot user studies was consistently positive.

Finally, Figure 9 illustrates the effect of the amount of training data on the performance of our approach. As expected, our model performs best when all training data is provided. However, we notice that the values of precision, recall, and F_1-score for lower amounts of training data are quite similar to the values obtained when using the full dataset. This indicates that our model is quite robust, as it can be effective with less data. Further interpreting the results, we may note that the dataset is annotated effectively, with few ambiguities, even when little training data is used for classifying concepts.

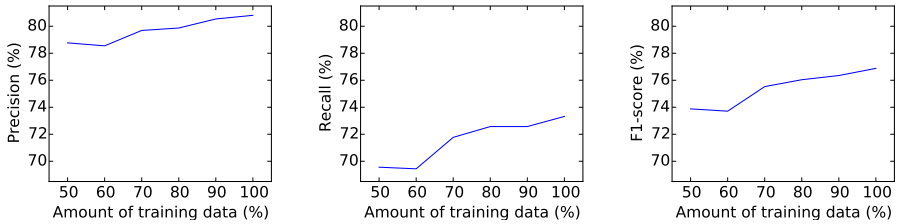


Fig. 9 Evaluation diagrams depicting (a) Precision, (b) Recall, and (c) F_1-score values for our model with different amount of training data used.

8 Conclusions

We conclude by summarising the main contributions of our work. To begin with, we collected a range of functional requirements and devised an ontology to describe the static view of software systems. With several hundreds of real-world examples, our functional requirements corpus provides a valuable starting point for the linguistic analysis of requirement texts. In addition to providing raw examples, our initial annotation effort led to more than a thousand annotations that showcase instances of the ontology. To the best of our knowledge, this is the first annotated resource of software requirements that is publicly available for research purposes.

As part of the annotation process, we defined an hierarchical annotation scheme in which formal representations are derived through step-wise refinement. Although our initial annotation effort only takes into account a subset of all concepts defined in the ontology, we observed a substantial level of agreement between annotators. Given the diverse source of requirements in our

collection, we take this result as an indicator that both the ontology and the annotation scheme are reliable and generic enough to be applied across various software application domains. Further work may be oriented towards integrating more concepts into the annotation scheme, possibly aided by providing more detailed guidelines to annotators.

The main aim of computer-assisted software engineering is to semi-automate the process of getting from software requirements to actual implementations. The ontology and annotations developed in this project form an essential building block towards this goal in that they provide a meaningful and structured representation of a software component. To truly assist software-engineering, however, the mapping from requirements to ontology instances needs to be performed in an automated manner. We therefore developed a semantic parsing model that automatically induces ontology-based representations from text. Our model achieves a high precision on this task and significantly outperforms two statistical baselines: one that learns word-level patterns and one that learns syntax-based patterns.

Given that requirements described in natural language are inherently prone to ambiguity, incompleteness and inconsistency, computational models will rarely achieve perfect performance. To allow users to revise potentially erroneous predictions by the model, we developed an easy-to-use annotation tool that implements the first step of our iterative annotation scheme as part of a web-based application. This tool has so far only been used internally within our project, but will be made available to any software developers using the S-CASE platform.

In future work, we will assess how our tool can improve the time and cost of the software development process, as measured, for example, in terms of person-hours. Furthermore, we will empirically validate the usefulness of our ontology, annotation scheme and automatic parser in a question answering scenario. That is, developers will be able to search for implemented software components in the S-CASE repository, using a query mechanism based on structured semantic representations.

Acknowledgements

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission.

References

Abbott RJ (1983) Program design by informal English descriptions. *Communications of the ACM* 26(11):882–894

Bach N, Badaskar S (2007) A Review of Relation Extraction, language Technologies Institute, Carnegie Mellon University

Bengio Y, Ducharme R, Vincent P, Jauvin C (2003) A neural probabilistic language model. *Journal of Machine Learning Research* 3:1137–1155

Berant J, Chou A, Frostig R, Liang P (2013) Semantic parsing on freebase from question-answer pairs. In: Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, pp 1533–1544

Björkelund A, Hafdell L, Nugues P (2009) Multilingual semantic role labeling. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task, Association for Computational Linguistics, Stroudsburg, PA, USA, CoNLL '09, pp 43–48

Boehm B, Basili VR (2001) Software defect reduction top 10 list. *Computer* 34:135–137

Bohnet B (2010) Top accuracy and fast dependency parsing is not a contradiction. In: Proceedings of the 23rd International Conference on Computational Linguistics, Beijing, China, pp 89–97

Bollacker K, Evans C, Paritosh P, Sturge T, Taylor J (2008) Freebase: A collaboratively created graph database for structuring human knowledge. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, New York, USA, SIGMOD '08, pp 1247–1250

Booch G (1986) Object-oriented development. *IEEE Transactions on Software Engineering* 12(2):211–221

Bunescu R, Mooney RJ (2005a) Subsequence kernels for relation extraction. In: Advances in Neural Information Processing Systems, Vol. 18: Proceedings of the 2005 Conference (NIPS), pp 171–178

Bunescu RC, Mooney RJ (2005b) A shortest path dependency kernel for relation extraction. In: Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Stroudsburg, PA, USA, pp 724–731

Cai Q, Yates A (2013) Large-scale semantic parsing via schema matching and lexicon extension. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Sofia, Bulgaria, pp 423–433

Clark S, Curran JR (2007) Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics* 33(4):493–552

Culotta A, Sorensen J (2004) Dependency tree kernels for relation extraction. In: Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, Stroudsburg, PA, USA, ACL '04, pp 423–429

Denger C, Berry DM, Kamsties E (2003) Higher quality requirements specifications through natural language patterns. In: Proceedings of the IEEE International Conference on Software: Science, Technology and Engineering, pp 80–90

Fan RE, Chang KW, Hsieh CJ, Wang XR, Lin CJ (2008) LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research* 9:1871–1874

Fleiss JL, Levin B, Paik MC (1981) The measurement of interrater agreement. *Statistical Methods for Rates and Proportions* 2:212–236

Gervasi V, Zowghi D (2005) Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering and*

Methodology 14(3):277–330

Ghosh S, Elenius D, Li W, Lincoln P, Shankar N, Steiner W (2014) Automatically extracting requirements specifications from natural language. *arXiv preprint arXiv:14033142*

Gildea D, Jurafsky D (2002) Automatic labeling of semantic roles. *Computational Linguistics* 28(3):245–288

Gordon M, Harel D (2009) Generating executable scenarios from natural language. *Computational Linguistics and Intelligent Text Processing* pp 456–467

GuoDong Z, Jian S, Jie Z, Min Z (2005) Exploring various knowledge in relation extraction. In: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, Stroudsburg, PA, USA, ACL '05, pp 427–434

Hajič J, Ciaramita M, Johansson R, Kawahara D, Martí MA, Màrquez L, Meyers A, Nivre J, Padó S, Štěpánek J, et al (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, pp 1–18

Harmain HM, Gaizauskas R (2003) Cm-builder: A natural language-based case tool for object-oriented analysis. *Automated Software Engineering* 10(2):157–181

Kambhatla N (2004) Combining lexical, syntactic, and semantic features with maximum entropy models for extracting relations. In: Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions, ACL, Stroudsburg, PA, USA, ACLdemo '04, pp 178–181

Kof L (2004) Natural language processing for requirements engineering: Applicability to large requirements documents. In: Workshop Proceedings of the 19th International Conference on Automated Software Engineering

Konrad S (2005) Facilitating the construction of specification pattern-based properties. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering, pp 329–338

Kwiatkowksi T, Zettlemoyer L, Goldwater S, Steedman M (2010) Inducing probabilistic CCG grammars from logical form with higher-order unification. In: Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, Cambridge, Massachusetts, pp 1223–1233

van Lamsweerde A (2009) *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley

Meyers A, Reeves R, Macleod C (2008) *NomBank v1.0*. Linguistic Data Consortium, Philadelphia

Mich L (1996) NL-OOPS: From natural language to object oriented requirements using the natural language processing system LOLITA. *Natural Language Engineering* 2(2):161–187

Mich L, Mariangela F, Pierluigi NI (2004) Market research for requirements analysis using linguistic tools. *Requirements Engineering* 9(1):40–56

Nanduri S, Rugaber S (1995) Requirements validation via automated natural language parsing. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, vol 3, pp 362–368

Palmer M, Gildea D, Kingsbury P (2005) The Proposition bank: An annotated corpus of semantic roles. *Computational Linguistics* 31(1):71–106

Post A, Hoenicke H (2012) Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In: Proceedings of the Fourth International Conference on Verified Software: Theories, Tools, and Experiments, pp 225–240

Pradhan SS, Ward WH, Hacioglu K, Martin JH, Jurafsky D (2004) Shallow semantic parsing using support vector machines. In: Susan Dumais DM, Roukos S (eds) HLT-NAACL 2004: Main Proceedings, Association for Computational Linguistics, Boston, Massachusetts, USA, pp 233–240

Roth M, Klein E (2015) Parsing software requirements with an ontology-based semantic role labeler. In: Proceedings of the IWCS Workshop Language and Ontologies 2015, pp 15–21

Roth M, Woodsend K (2014) Composition of word representations improves semantic role labelling. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, Doha, Qatar, pp 407–413

Roth M, Diamantopoulos T, Klein E, Symeonidis A (2014) Software requirements: A new domain for semantic parsers. In: Proceedings of the ACL 2014 Workshop on Semantic Parsing, Baltimore, Maryland, USA, pp 50–54

Saeki M, Horai H, Enomoto H (1989) Software development process from natural language specification. In: Proceedings of the 11th International Conference on Software Engineering, pp 64–73

Steedman M (2000) *The Syntactic Process*, vol 35. MIT Press

Tang LR (2003) Integrating top-down and bottom-up approaches in inductive logic programming: Applications in natural language processing and relational data mining. PhD thesis, Department of Computer Sciences, University of Texas, Austin, Texas, USA

Tjong SF, Hallam N, Hartley M (2006) Improving the quality of natural language requirements specifications through natural language requirements patterns. In: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology, Washington, DC, USA, pp 199–205

Wong YW, Mooney RJ (2006) Learning for semantic parsing with statistical machine translation. In: Proceedings of the Main Conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, Association for Computational Linguistics, Stroudsburg, PA, USA, HLT-NAACL '06, pp 439–446

Yeh A (2000) More accurate tests for the statistical significance of result differences. In: Proceedings of the 18th International Conference on Computational Linguistics, Saarbrücken, Germany, pp 947–953

Zelenko D, Aone C, Richardella A (2003) Kernel methods for relation extraction. *J Mach Learn Res* 3:1083–1106

Zettlemoyer L, Collins M (2007) Online learning of relaxed CCG grammars for parsing to logical form. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, Prague, Czech Republic, pp 678–687

Zhao S, Grishman R (2005) Extracting relations with integrated information using kernel methods. In: Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics, Association for Computational Linguistics, Stroudsburg, PA, USA, ACL '05, pp 419–426

Zolotas C, Diamantopoulos T, Chatzidimitriou KC, Symeonidis AL (2016) From Requirements to Source Code: a Model-Driven Engineering approach for RESTful Web Services. *Automated Software Engineering* In press