Software Requirements: A new Domain for Semantic Parsers

Michael Roth† **Themistoklis Diamantopoulos**‡ **Ewan Klein**† **Andreas Symeonidis**‡
†ILCC, School of Informatics
University of Edinburgh
{mroth,ewan}@inf.ed.ac.uk
‡Electrical & Computer Engineering Department
Aristotle University of Thessaloniki
thdiaman@issel.ee.auth.gr
asymeon@eng.auth.gr

Abstract

Software requirements are commonly written in natural language, making them prone to ambiguity, incompleteness and inconsistency. By converting requirements to formal semantic representations, emerging problems can be detected at an early stage of the development process, thus reducing the number of ensuing errors and the development costs. In this paper, we treat the mapping from requirements to formal representations as a semantic parsing task. We describe a novel data set for this task that involves two contributions: first, we establish an ontology for formally representing requirements; and second, we introduce an iterative annotation scheme, in which formal representations are derived through step-wise refinements.

1 Introduction

During the process of software development, developers and customers typically discuss and agree on requirements that specify the functionality of a system that is being developed.[1] Such requirements play a crucial role in the development lifecycle, as they form the basis for actual implementations, corresponding work plans, cost estimations and follow-up directives (van Lamsweerde, 2009). In general, software requirements can be expressed in various different ways, including the use of UML diagrams and storyboards. Most commonly, however, expectations are expressed in natural language (Mich et al., 2004), as shown in Example (1):

(1) A user should be able to login to his account.

[1]Although software engineering can also involve *non-functional* requirements, which describe general quality criteria of a system, this paper is only concerned with functional requirements, i.e., requirements that specify the behavior of a system.

While requirements expressed in natural language have the advantage of being intelligible to both clients and developers, they can of course also be ambiguous, vague and incomplete. Although formal languages could be used as an alternative that eliminates some of these problems, customers are rarely equipped with the mathematical and technical expertise for understanding highly formalised requirements. To benefit from the advantages of both natural language and formal representations, we propose to induce the latter automatically from text in a semantic parsing task. Given the software requirement in Example (1), for instance, we would like to construct a representation that explicitly specifies the types of the entities involved (e.g., *object*(account)) and that captures explicit and inferable relationships among them (e.g., *owns*(user, account)). We expect such formal representations to be helpful in detecting 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 and hence more expensive stage (Boehm and Basili, 2001).

Given the benefits of formal representations, we believe that software requirements constitute a useful application domain for semantic parsers. Requirement texts naturally occur in the real world and appropriate data sets can thus be constructed without setting up artificial tasks to collect them. Parsing requirements of different software projects also poses interesting challenges as texts exhibit a considerable amount of lexical variety, while frequently also containing more than one relation per sentence.

2 Related Work

A range of methods have been proposed in previous work to (semi-)automatically process requirements written in plain, natural language text and map them to formal representations. To the best of our knowledge, Abbott (1983) was the first to introduce a technique for extracting data types, variables and operators from informal texts describing a problem. 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 of similar complexity that extends Abbot's approach to object-oriented development. Saeki et al. (1989) implemented a first prototype that automatically constructs object-oriented models from informal requirements. As proposed by Abbott and Booch, the system is based on automatically extracted nouns and verbs. Although Saeki et al. 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 requirement texts by applying a syntactic parser and a set of post-processing rules. In a similar setting, Mich (1996) employed a full NLP pipeline that contains a semantic analysis module, thus omitting the need for additional post-processing rules. More recent approaches include those by Harmain and Gaizauskas (2003) and Kof (2004), who relied on a combination of NLP components and human interaction. Whereas most approaches in previous work aim to derive class diagrams, Ghosh et al. (2014) proposed a pipeline architecture that converts syntactic parses to logical expressions via a set of heuristic post-processing rules.

Despite this seemingly long tradition, previous methods for processing software requirements have tended to depend on domain-specific heuristics and knowledge bases or have required additional user intervention. In contrast, we propose to utilize annotated data to learn how to perform semantic parsing of requirements automatically.

3 Data Set

Given our conviction that mapping natural language software requirements to formal representations provides an attractive challenge for semantic parsing research, we believe that there is a more general benefit in building a corpus of annotated requirements. One immediate obstacle is that software requirements can drastically differ in quality, style and granularity. To cover a range of possible differences, we asked lecturers from several universities to provide requirement documents written by students. We received requirement documents on student projects from various domains, including embedded systems, virtual reality and web applications.[2] From these documents, we extracted lists of requirements, each of which is expressed within a single sentence. We additionally collected single sentence requirements within the S-CASE project, describing industrial prototypes of cloud-based web services.[3] Table 1 gives an overview of the quantity of requirements collected. We observe that the number of requirements received for student projects is much higher. 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 to be more regimented and stylised than those written by senior software engineers. Examples (2) and (3) show examples of a student-written and developer-written requirement, respectively.

(2) The user must be able to vote on polls.

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

In comparison to two extant data sets, namely GeoQuery880 (Tang, 2003) and Free917 (Cai and Yates, 2013), we find that our collection is still relatively small in terms of example sentences. The

	#sentences	#tokens	#types
student projects	270	3130	604
industrial prototypes	55	927	286
Our dataset (total)	325	4057	765
GEOQUERY880	880	6656	279
FREE917	917	6769	2035

Table 1: Statistics on our requirements collection and existing semantic parsing data sets.

[2]The majority of collected requirements are from a software development course organized jointly by several European universities, cf. http://www.fer.unizg.hr/rasip/dsd

[3]http://www.scasefp7.eu/

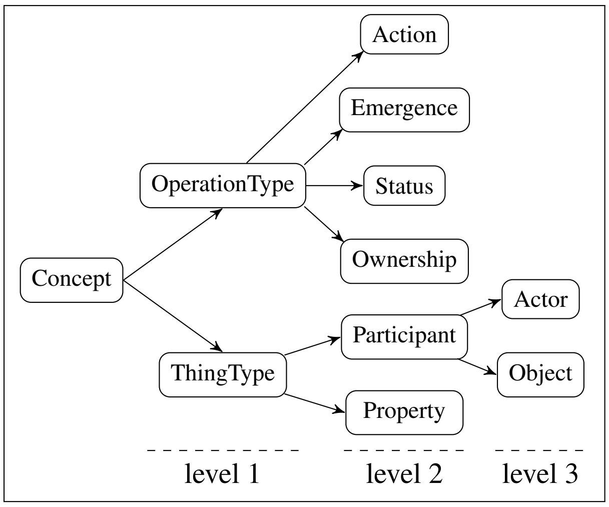

Figure 1: Class hierarchy of our conceptual ontology for modeling software requirements.

difference in total number of tokens is not as crucial, however, given that sentences in our data set are much longer on average. We further observe that the token/type ratio in our texts lies somewhere between ratios reported in previous work. Based on the observed lexical variety and average sentence length, we expect our texts to be challenging but not too difficult to parse using existing methods.

4 Modeling Requirements Conceptually

Different representations have been proposed for modeling requirements in previous work: whereas early work focused on deriving simple class diagrams, more recent approaches suggest representing requirements via logical forms (cf. Section 2).

In this paper, we propose to model requirements using a formal ontology that captures general concepts from different application domains. Our proposed ontology covers the same properties as earlier work and provides a means to represent requirements in logical form. In practice, such logical forms can be induced by semantic parsers and in subsequent steps be utilized for automatic inference. The class hierarchy of our ontology is shown in Figure 1. At the highest level of the class hierarchy, we distinguish between "things" (*ThingType*) and "operations" (*OperationType*).

4.1 ThingType

We define the following subclasses of *ThingType*:

- A *Participant* is a thing that is involved in an operation. We further subdivide *Participant*s into *Actor*s, which can be users of a system or the system itself, and *Object*s.
- A *Property* is an attribute of an *Object* or a characteristic of an *OperationType*.

4.2 OperationType

We further divide operations into the following subclasses:

- An *Action* describes an operation that is performed by an *Actor* on one or several *Object*(s).
- A *State* is an operation that describes the status of an *Actor*.
- *Ownership* is used to model operations that express possession.
- *Emergence* represent operations that undergo passive transformation.

4.3 Relations

In addition to the class hierarchy, we define a set of relations between classes, which describe and constrain how different operations and things can interact with each other.

On the level of *OperationType*, every operation can be assigned one *Actor* via the relations HAS_ACTOR or HAS_OWNER, respectively. *Object*s can participate in *Action*s, *State*s and *Ownership*s via the relations ACTS_ON, HAS_STATE and OWNS, respectively. Every instance of *OperationType* and *Object* can further have an arbitrary number of properties assigned to it via the relation HAS_PROPERTY.

5 Annotation Process

In preliminary annotation experiments, we found that class diagrams may be too simple to represent requirements conceptually. Logical forms, on the other hand, can be difficult to use for annotators without sufficient background knowledge. To keep the same level of expressiveness as logical forms and the simplicity of object-oriented annotations, 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 4, we can naturally divide each annotation iteration according to a level in the ontology. This means that in the first iteration, we ask annotators

A user that is logged in to his account must be able to update his password.		
Actor(user)	∧ *Action*(login)	∧ *Action*(update)
∧ *Object*(account)	∧ HAS_ACTOR(login,user)	∧ HAS_ACTOR(update,user)
∧ *Object*(password)	∧ ACTS_ON(login,account)	∧ ACTS_ON(update,password)
	∧ *Ownership*(o_1)	∧ *Ownership*(o_2)
	∧ HAS_OWNER(o_1,user)	∧ HAS_OWNER(o_2,user)
	∧ OWNS(o_1,account)	∧ OWNS(o_2,password)
The system must be able to forward and rewind a playing program.		
Actor(system)	∧ *Action*(forward)	∧ *Action*(rewind)
∧ *Object*(program)	∧ HAS_ACTOR(forward,system)	∧ HAS_ACTOR(rewind,system)
	∧ ACTS_ON(forward,program)	∧ ACTS_ON(rewind,program)
	∧ *Property*(playing)	∧ HAS_PROPERTY(program,playing)

Table 2: Example requirements from different domains and logical forms derived from annotations.

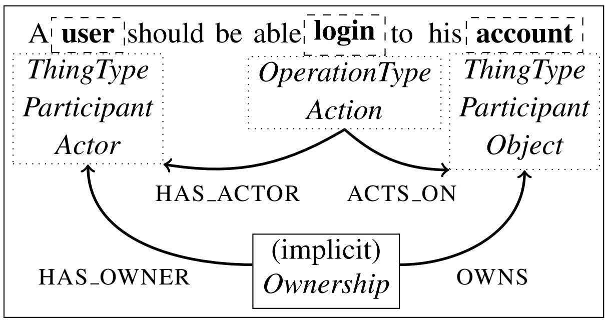

Figure 2: Annotation process: instances are marked in text (dashed), class assignments are refined (dotted), and relations are added (solid).

to simply mark all instances of *ThingType* and *OperationType* that are explicitly expressed in a given requirement. We then resolve conflicting annotations and present the resulting instances from the first level to annotators for the next iteration. In each iteration, we add one layer of sophistication from the class hierarchy, resulting in step-wise refinements. In the final iteration, we add relations between instances of concepts, including implicit but inferable cases.

An illustration of the overall annotation process, based on Example (1), is depicted in Figure 2. The last iteration in this example involves the addition of an *Ownership* instance that is indicated (by the phrase "his account") but not explicitly realized in text. Although identifying and annotating such instances can be more challenging than the previous annotation steps, we can directly populate our ontology at this stage (e.g., via conversion to RDF tuples) and run verification tools to check whether they are consistent with the annotation schema.

6 Discussion

The annotation scheme introduced in Section 4 is designed with the goal of covering a wide range of different application domains. Although this means that many of the more fine-grained distinctions within a domain are not considered here, we believe that the scheme already provides sufficient information for a range of tasks. By storing processed requirements in a relational database, for example, they can be retrieved using structured queries and utilized for probabilistic inference.

Given the hierarchical structure of our annotation process, as defined in Section 5, it is possible to extend existing annotations with additional levels of granularity provided by domain ontologies. As an example, we have defined a domain ontology for web services, which contains subclasses of *Action* to further distinguish between the HTTP methods *get*, *put*, *post* and *delete*. Similar extensions can be defined for other domains.

Regarding the task of semantic parsing itself, we are currently in the process of annotating several hundreds of instances of requirements (cf. Section 3) following the proposed ontology. We will release an initial version of this data set at the Semantic Parsing workshop. The initial release will serve as a basis for training and evaluating parsers in this domain, for which we are also planning to collect more examples throughout the year. We believe that requirements form an interesting domain for the parsing community as the texts involve a fair amount of variation and challenging semantic phenomena (such as inferable relations), while also serving a practical and valuable purpose.

Acknowledgements

Parts of this work have been supported by the FP7 Collaborative Project S-CASE (Grant Agreement No 610717), funded by the European Commission. We thank our project partners for data support and useful discussions on the proposed ontology.

References

Russell J Abbott. 1983. Program design by informal english descriptions. *Communications of the ACM*, 26(11):882–894.

Barry Boehm and Victor R. Basili. 2001. Software defect reduction top 10 list. *Computer*, 34:135–137.

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

Qingqing Cai and Alexander Yates. 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)*, pages 423–433, Sofia, Bulgaria, August.

Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. 2014. Automatically extracting requirements specifications from natural language. *arXiv preprint arXiv:1403.3142*.

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

Leonid Kof. 2004. Natural language processing for requirements engineering: Applicability to large requirements documents. In *19th International Conference on Automated Software Engineering, Workshop Proceedings*.

Luisa Mich, Franch Mariangela, and Novi Inverardi Pierluigi. 2004. Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1):40–56.

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

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.

Motoshi Saeki, Hisayuki Horai, and Hajime Enomoto. 1989. Software development process from natural language specification. In *Proceedings of the 11th International Conference on Software Engineering*, pages 64–73.

Lappoon R. Tang. 2003. *Integrating Top-down and Bottom-up Approaches in Inductive Logic Programming: Applications in Natural Language Processing and Relational Data Mining*. Ph.D. thesis, Department of Computer Sciences, University of Texas, Austin, Texas, USA, August.

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