Automated Software Engineering (2024) 31:5
https://doi.org/10.1007/s10515-023-00403-y

A tool for requirements engineering using ontologies and boilerplates

Christina Antoniou[1] · Nick Bassiliades[1]

Received: 13 April 2023 / Accepted: 28 October 2023

Abstract

The most popular technique for specification requirements is natural language. The disadvantage of natural language is ambiguity. Boilerplates are syntactic patterns which limit the ambiguity problem associated with using natural language to specify system/software requirements. Also, using boilerplates is considered a useful tool for inexperienced engineers to define requirements. Using linguistic boilerplates, constrains the natural language syntactically. Furthermore, a domain-specific ontology is used to constrain requirements semantically, as well. In requirements specification, using ontologies helps to restrict the vocabulary to entities, properties, and property relationships which are semantically related. The above results in avoiding or making fewer mistakes. This work makes use of the combination of boilerplate and ontology. Usually, the attributes of boilerplates are completed with the help of the ontology. The contribution of this paper is that the whole boilerplates is stored in the ontology and attributes and fixed elements are part of the ontology. This combination helps to correct semantically and syntactically requirement construction. This paper proposes a tool based on a domain-specific ontology and a set of predefined generic linguistic boilerplates for requirements engineering. We create a domain-specific ontology and a minimal set of boilerplates for an ATM (Automated Teller Machine). We carried out an experiment in order to obtain evidence for the effectiveness and efficiency of our method. The experiment took the form of a case study for the ATM domain and our proposed method was evaluated by users. The contribution and novelty of our methodology is that we created a tool for defining requirements that integrates boilerplate templates and an ontology. We exploit the boilerplate language syntax, mapping them to Resource Description Framework triples which have also a linguistic nature.

Keywords Boilerplates · Requirements specification · Ontology

✉ Christina Antoniou
antoniouc@csd.auth.gr

[1] School of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece

1 Introduction

In order to successfully implement the development of a system, it is necessary to determine the requirements of the system and to document them with the appropriate technique. The documentation technique facilitates the communication between stakeholders. It also improves the quality of requirements. Documentation of requirements for a system can be done in three ways: requirements documentation using natural language, requirements documentation using conceptual models and hybrid requirements documents (Pohl and Rupp 2011).

Natural language, specifically prose, is considered the most well-known technique for documentation. The most important advantage is that the stakeholders do not need to learn anything new to use this technique. Also, the requirements engineer can use this method to document any kind of requirement (Pohl and Rupp 2011). But the downside of this technique is ambiguity (Arora et al. 2013, 2015).

On the other hand, conceptual models cannot be used for all kinds of requirements as the documentation using natural language. Nevertheless, conceptual models deal with the ambiguity of natural language. Hybrid requirements documents is the third way of recording and a combination of the two above ways. The third way takes advantage of the advantages of both and minimizes the disadvantages of both techniques (Pohl and Rupp 2011).

Requirement templates is an approach for constructing requirements which uses templates and glossaries. This way of documenting requirements is not difficult and helps to limit the disadvantages from the documentation using natural language. Pohl and Rupp, state the following definition for the requirement template "*a requirement template is a blueprint for the syntactic structure of individual requirements*." The definition mentioned by Pohl and Rupp actually refers to boilerplates. The boilerplates are also called requirement templates (Pohl and Rupp 2011).

Pohl and Rupp (2011) mention a process which consists of 5 steps in order to create the requirement template. In the first step, the degree of legal obligation for the requirement is determined. There are the following categories of the obligation of requirements: legally obligatory, urgently recommended, future, and desirable. The second step is the core of the requirement. In this step, the functionality of the requirement is determined with $<$*process*$>$, for example the system store, prints etc. The third step characterizes the activity of the system in three following categories: (a) autonomous system activity, for example *the process is performed autonomously by the system* (process verb), (b) user interaction, e.g., the process is provided as a service for the user by the system (provide), and (c) the interface requirement, e.g., an external event triggers the system to execute the process (be able to), i.e., the system is waiting for a message or data to react. In the fourth step, the process-verbs are completed with objects. Some verbs can have more than one object. In the fifth step, logical and temporal conditions are determined. The system executes processes under these conditions. Figure 1 shows the diagram of the boilerplates of Pohl and Rupp.

Other popular types of boilerplates are EARS templates or boilerplates (Mavin et al. 2009). The EARS boilerplates are distinguished in (a) ubiquitous requirements:

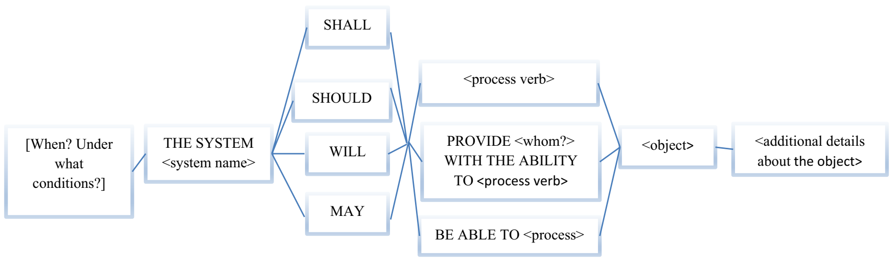

[When? Under what conditions?]
THE SYSTEM <system name>
SHALL
SHOULD
WILL
MAY
<process verb>
PROVIDE <whom?> WITH THE ABILITY TO <process verb>
BE ABLE TO <process>
<object>
<additional details about the object>

Fig. 1 Boilerplates of Pohl and Rupp

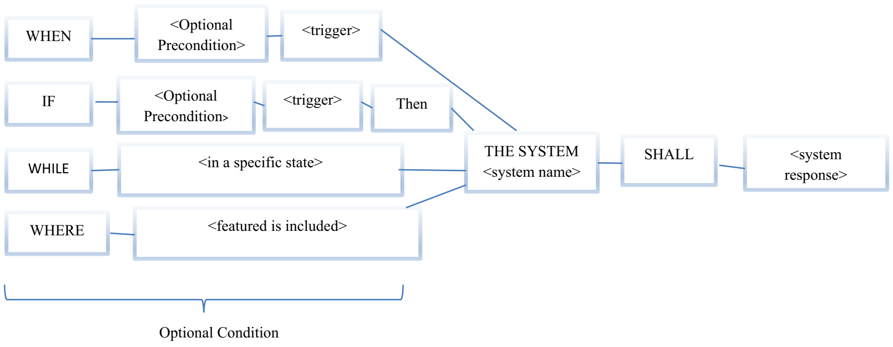


Fig. 2 EARS Boilerplates

these constitute the simplest structure (b) event-driven requirements: these are triggered when an event is detected, (c) unwanted behavior requirements (if then) handle unwanted situations, (d) state-driven requirements (while) are used for a specified state, or (e) optional feature requirements (where) are used when a certain feature appears. Figure 2 shows the diagram of boilerplates by Arora et al., (2014).

Hull et al., (2010) called requirements templates as boilerplates. Boilerplates are an easy and simple approach to standardize the natural language. To formulate a requirement with a boilerplate, it is necessary to select the appropriate boilerplate from a collection and fill in the blanks (attributes) with data. To understand the above, suppose there is a collection of templates like the following: "*The < system > shall be able to < function > < object > of type < qualification > within < performance > < units > ", The < system > shall be able to < function > < object > not less than < performance > times per < units > ", The < system > shall < function > < object > every < performance > < units >* ". A requirement engineer chooses the appropriate template and fills in the words in angle brackets with data. For example, the last template can become as follows: *The < coffee machine > shall < produce > < a cold drink > every < 5 > < seconds >* ". A boilerplate consists of some fixed elements and some attributes that the requirements engineer completes in order to create a requirement (as shown in Table 1).

Table 1 Examples of fixed and attributes elements of boilerplate

Boilerplate	Fixed elements	Attributes elements	Example completed Boilerplate
The < system > shall be able to < function > < object > of type < qualification > within < performance > < units > Hull et al., (2010)	*The, shall be able to, of type, within*	*< system >, < function >, < object >, < qualification >, < units >*	*The < coffee machine > shall < produce > < a cold drink > every < 5 > < seconds >*
The < system name > shall < system response > (Mavin et al. 2009)	*The, shall*	*< system name >, < system response >*	*The control system shall prevent engine overspeed*
< subject > < verb > < object > Our boilerplate		*< subject > < verb > < object >*	*< ATM > < returns > < CashCard >*
< subject > < sends > < object > To < entity > Our boilerplate	*To*	*< subject > < sends > < object > < entity >*	*< ATM > sends < typedpassword > of < Customer > To < BankComputer >*

The advantages of boilerplates are the following: (a) changes related to the expression of requirements affect only the corresponding boilerplate (Hull et al. 2010), (b) the processing of system information is very easy (for example to filter and sort a specific attribute) (Hull et al. 2010), (c) important information may be hidden (Hull et al. 2010), (d) flexibility can be maintained even in a small number of boilerplates (Farfeleder, Moser, Krall, Stålhane, Omoronyia & Zojer, 2011), (e) they are useful for the inexperienced engineers (Daramola, Sindre & Moser, 2012), (f) improve the creation of high quality requirements (Farfeleder, Moser, Krall, Stålhane, Zojer & Panis, 2011; Anuar et al. 2015), (g) reduce the natural effects such as ambiguity (Mahmud et al. 2015), (h) they are reusable (Ibrahim et al. 2009; Mahmud et al. 2016), (i) it is a simple and easy to understand method and you don't need to learn anything new to use them (Warnier & Condamines 2017), (j) new boilerplates can be created from the existing ones (Daramola et al. 2011; Do et al. 2020), (k) a repository can be created which will be updated with new categories, which is useful for the inexperienced engineers (Daramola et al. 2011), (l) help in creation to requirement models for verification (Zichler & Helke 2019), and finally, (m) they are also used in formalization (Zaki-Ismail et al. 2020) (n) they create coherent and concise requirement sentences (Haris & Kurniawan 2020).

Another way of documenting requirements is the formal specification. Due to its high degree of formality, it addresses the difficulties arising from the use of natural language (Pohl and Rupp 2011). Mahmud et al., (2015) report that the formal specification is "*… the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy*".

Well-written requirements are created from the combination of boilerplates and ontologies (Fanmuy et al. 2012). Also, requirements specification effort for the security domain is reduced because of the combination of ontology and boilerplates (Daramola, Sindre & Moser, 2012; Daramola, Sindre & Stalhane 2012). This combination seems useful to non-experienced engineers and the quality of requirements is improved. Gap filling can be done with the help of ontology as well as the combination helps to discover relationships between requirements (Daramola, Sindre & Stalhane, 2012).

In many cases, as we have seen so far, the combination of boilerplates with the ontology is used to fill in the gaps from the ontology. Although there are tools and methods such as Too et al., (2022), Kravari et al., (2021), Mokos & Katsaros (2020), Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Omoronyia, I., & Zojer, H. (2011), Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., & Panis, C. (2011) that make use of the advantages offered by combining boilerplates with the ontology for the purpose of defining requirements. However, these works make use of the ontology to fill only the attributes of boilerplates by ontology. Our contribution, which constitutes an innovation, the whole boilerplate is stored in the ontology. More specifically, both fixed elements and attributes are part of the ontology. Our method built into the tool leads the user to semantically correct requirement construction.

In this paper, we present the use of boilerplates with the ontology. The boilerplates exploit the linguistic nature of Resource Description Framework (RDF) (Richard et al., 2014) triples and create the boilerplate syntax. The use of both ontology and boilerplate help to limit the ambiguity caused by natural language. RDF describes resources and the relationships between them in triplet form. It is a data model that contains statements such as *subject—predicate (verb)—object* much alike the boilerplate language. The classification of our boilerplates is: a) basic boilerplate template and extended form of basic boilerplate template and b) and boilerplate template with temporal or logical conditions. The basic boilerplate template consists of *subject verb object*, much the same as an RDF triple. The complex boilerplate template consists of a small number of RDF triples (a small semantic graph). Also, we develop a tool in which the engineer creates boilerplates which are based on Resource Description Framework triples. The user enters the requirement and selects the appropriate boilerplate template. The tool with the help of natural language processing and a domain ontology, suggests the appropriate values for attributes to the user. The purpose of this research is the development of a tool for constructing requirements based on the proposed combination of ontology and boilerplates. Our method incorporated into the tool aims to alleviate from the semantical ambiguity associated with natural language. Finally, we conduct an experiment in the form of a case study with users in order to evaluate our method.

The rest of the paper is organized as follows. In Sect. 2, we first briefly present related work. In Sect. 3, we present the boilerplates based on the linguistic nature of RDF triples that is similar to the syntax of the boilerplates. Also, we present and discuss the domain ontology. Section 4 introduces our tool based on ontology and natural language processing guidance. Section 5 presents the evaluation of our method. Finally, we present the conclusions in Sect. 4.

2 Related work

Pasquariello et al., (2022) propose a framework using boilerplates for syntactically correct requirements specification. The purpose is the quality of the requirements, which can be verified promptly on time based on predefined criteria such as clarity, singularity, conformity, and descriptiveness. Also, they implemented a requirement tool which helps the user to create correct requirements during construction.

Too et al., (2022) suggest a requirement tool which is called UReST. This tool was created to enhance and assist the engineer in requirement engineering (RE) activities. The combination of ontology and boilerplates was incorporated in UReST. This approach support and guides the engineer in order to create the requirements.

Fritz et al., extract information automatically from text-based requirements. Initially, with the help of a database, the requirements are distinguished from non-requirements as long as the requirement document is transferred with little effort to the requirements database. Then, the necessary features or components of the system, which are recorded in the document, are detected. This method helps to clarify the requirements that are relevant for placeholders. Completeness of requirements is achieved by extracting semantic roles and using boilerplates in order to reduce errors (Fritz et al. 2021).

Kravari et al., (2020) state that for the implementation of a system an important step is the documenting of the requirements. By extension, the success or the failure in the development of a system largely depends on well-defined specifications.

Semantics, natural language processing, ontology and boilerplates are included in the new approach that they propose. Furthermore, Kravari et al., (2021) in order to generate high-quality requirements, they developed a tool, which is called SENSE, for constructing requirements. This tool integrates template-language, specifically boilerplates, semantics, natural language processing and ontology. After processing a set of boilerplates, this framework suggests the appropriate boilerplate depending on the type of requirement. The requirement engineer can use the SENSE, which is considered a simple and easily understandable approach in order to construct well-defined requirements. Also, the tool can perform verification by using SPARQL (SPIN) queries.

The research of Mokos & Katsaros (2020) was related to formalization of requirements and their validation. Specifically, they mention that an ontology is used to requirements specification. Requirements describe a system, and requirements specifications can be implemented by mapping concepts to semantic roles. In order to avoid ambiguity, they mention as a solution the languages which are based on templates such as boilerplates. Also, they examined the derivation of formal properties from the requirements.

The work of Do et al., (2019) is about a new approach which promotes, launches creativity in software so that the software development companies become competitive. This approach can be used in both new and existing systems. Creative requirements are extracted with the help of reusable requirements, natural language processing, machine learning and boilerplates. Boilerplates play an important role in the novel framework.

Ahmad et al., (2018) proposed a tool which helps to improve the quality of Software Requirements Specification. This particular tool-based boilerplate was evaluated for the quality that offers in the terms of comprehensibility, correctness and consistency. The central idea behind the tool is to discover the basic, essential requirements for an information management system and turn them into statements about requirement specification.

The creation of high-level requirements of a system are important to avoid errors in subsequent stages as well as in the verification and validation phase. The most well-known technique for specification requirements is natural language. The disadvantage of natural language is ambiguity. Boilerplates reduce ambiguity and do not require special training to use them. So far, we have seen the combination of ontology and boilerplates for the specification of requirements. Regarding the creation or the use of boilerplates, the variants of Hull (Pasquariello et al., (2022); Daramola et al., (2011); Daramola, O., Sindre, G., & Moser, T. (2012)), EARS (Mavin et al. 2009; Arora et al. 2015) and Rupp and Pohl (Arora et al. 2015) boilerplates have been mainly used by the literature. We also notice in the above works (where there is a combination of boilerplate and ontology) that the completion of the attributes of boilerplate is done via an ontology.

In this work, we developed a tool based on the boilerplate language and ontology. The tool accepts the requirement from the engineer in natural language and the user selects the appropriate boilerplate according to the syntactic structure of the requirement. The options of attributes result from natural language processing and ontology. The syntax of boilerplate is *subject-verb-object*. Similarly, the RDF triples also follow the same syntax *subject-predicate (verb)-object*. We exploit the syntax of the boilerplate language, mapping them to RDF triples. This constitutes the contribution and novelty of this work. Therefore, not only the attributes of the boilerplates are completed by the ontology but the whole boilerplate and the corresponding fixed elements are part of the ontology. Taking into account this advantage, namely the linguistic nature of both, we integrated all of the boilerplates into the ontology and not just the blanks of attributes that the engineer filled. Generally, boilerplates have some fixed word and some attributes which are completed manually from engineer or with help of ontology. Our tool suggests options for attributes with the help of the ontology and via natural language processing. Finally, we conducted a user-based experiment in the form of a case study in order to evaluate our method.

3 Ontology and boilerplates

3.1 Ontology

In this article, we developed a tool that accepts a requirement and suggests appropriate attributes of boilerplates that correspond to the specific requirement typed by the user. For the appropriate suggestion for the user, the tool uses the help of natural language processing as well as the ontology we have created for the ATM use case. In this section we will discuss and present the ontology for the ATM use case.

Gruber mentions that an ontology is "an explicit specification of a conceptualization" (Gruber 1993, 1995). Antoniou et al., (2011) refer that the ontology is "a model of a particular domain built for a particular purpose". Guarino et al., (2009) define that ontologies are "a means to formally model the structure of a system". Ontology represents the knowledge of a domain i.e., concepts that exist in the real world and defines entities of the domain and relationships between entities of the domain. The cornerstone of an ontology rests on the hierarchy of concepts (Guarino et al. 2009).

Some basic components of an ontology are classes, attributes or data properties, relations or object properties, and individuals. RDF describes resources and the relationships between them in a triplet form (RDF triplet). It is a data model that contains statements such as *subject—predicate (or verb phrase)—object* much alike the boilerplate language, especially its most basic form, namely *subject-verb-object*. Class is a set of objects that share some common properties with the other members of the set. Data properties are attributes or properties of classes. Relations are relationships between classes of the knowledge domain. The domain of a property/predicate indicates the class of the subject in a triplet. The range of a property/predicate indicates the class of the object in a triplet. Individuals are instances of classes (Staab & Studer, 2009). We created an ontology for the ATM domain, namely classes, class hierarchies, object properties, data properties using the Protégé ontology editor (Musen 2015).

First, we mention the most basic concepts for the case study are the following: Account, ATM, Bank, BankComputer, CashCard, Customer, and Transaction. The bank has customers and offers products. One of the products that Bank offers is the

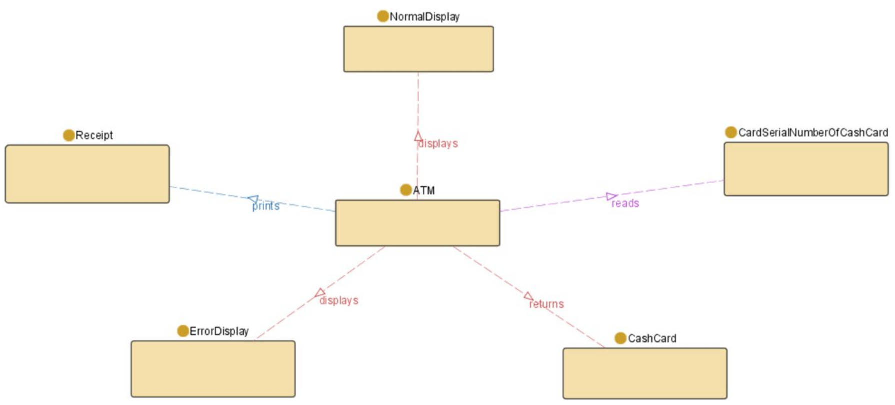


Fig. 3 Relationships among the main classes of the ontology

Account. The Account is used for the various transactions that the customer needs. It is possible for a customer to have more than one Account. Figure 3 shows the main classes of the ontology. Figure 3 shows examples of relationships among the main classes of the ontology, such as ATM returns CashCard, ATM reads CardSerialNumberOfCashCard, ATM displays NormalDisplay, ATM displays ErrorDisplay, ATM prints Receipt. The classes (Receipt, NormalDisplay, Receipt, CashCard, ErrorDisplay, CardSerial Number ofCashCard) are related to each other by object properties (displays, prints, returns, reads).

ATM is a service that enables customers to perform transactions. We defined the main ATM class in the ontology as a subclass of the class Service as shown in Fig. 3. Also, the ATM is the link between the customer and the bank computer (BankComputer). For this communication it is necessary for the customer to insert the cash card in order to make the necessary transactions. We set the CashCard class as subclaas of Card class and the Transaction entity is a top-level class, as shown in Fig. 4. Specifically, the Customer selects the service options through the ATM and these are transferred to the BankComputer which approves or rejects the customer's applications. For example, *the Customer requests a withdrawal of 50 euros*, the request is transferred to the BankComputer and if there are any available in the customer's Account, the money is made available to the Customer.

According to the above, there is an interaction between the customer and the ATM and also between the ATM and the bank computer. Basically, the interaction is related to the request made by the customer through the ATM and the response received from the bank computer through the ATM. For this reason, we created an interaction class that has two subclasses, *Request* and *Response* as shown in Fig. 4.

3.2 Boilerplates

We were inspired by the most popular boilerplates-templates such as the ones of Pohl and Rupp (2011), EARS (Mavin et al. 2009) and Hull et al., (2010). The common ground in the aforementioned boilerplates is the following syntax:

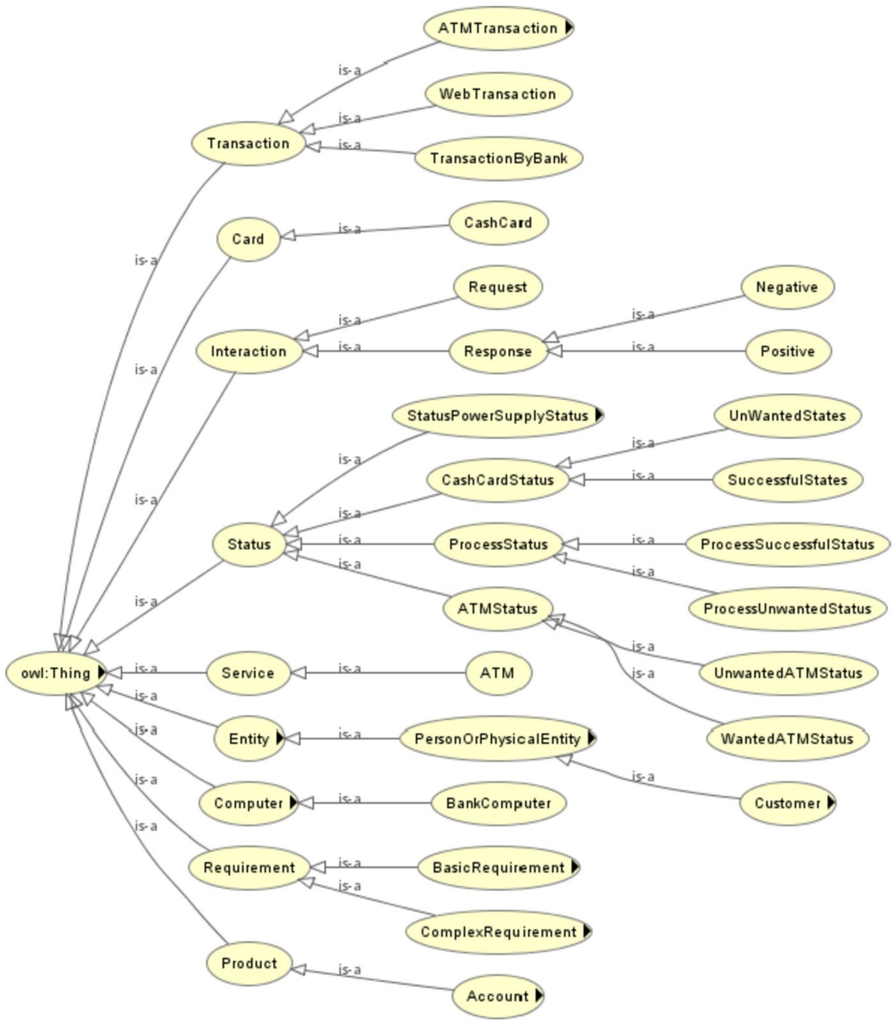


Fig. 4 Hierarchy of the main classes of the ontology

subject-verb-object. We exploited the boilerplate syntax which is similar to the linguistic nature syntax of RDF triples. Both RDF and boilerplates have a similar syntax, in the form of *subject-predicate-object* triples. Taking advantage of this syntax, we incorporated all of the boilerplates into the ontology and not just the blanks (attributes) to be filled. According to Hull et al., (2010) boilerplates can be extended and reused. Also, Hull et al., mention that when you have a requirement and look for the appropriate boilerplate from the collection and don't find it then a new boilerplate must be created. Boilerplates can be extended-adapted to fit the requirements.

In terms of compulsion of the requirements, it can be represented in the template as a detail, for example *<ATM>[shall]<returns><CashCard>*. Also, since we have integrated the whole boilerplate requirements into the ontology and not just the attributes, we represent the obligation of the requirement as a datatype property. The compulsion of the requirements is orthogonal to the rest of the representation. Each boilerplate instance (which is actually a requirement) has a compulsion datatype property whose value is one of the words used in the requirements standards (e.g., *shall*, *may*, etc.). The boilerplate syntax also includes these words as optional before the verb. Our tool then gives access to this compliance property to subsequent tools that will use the requirements stored in the ontology for further processing. The Table 2 below shows the similarities and differences of EARS, Pohl and Rupp with our boilerplates.

In terms of the methodology for our boilerplates, the requirements engineer can select values for the attributes of boilerplates, only those related to the domain/range of the properties, i.e. the verbs that connect the requirements. The methodology for the creation of boilerplates aimed to limit semantic errors during the design phase. That is why the introduction of entities, i.e. the completion of the attributes of boilerplates, was limited and is done according to those entities that are semantically related to the appropriate verbs (object properties). Some pre-existing boilerplates, such as EARS (Mavin et al. 2009) or Pohl and Rupp (Pohl and Rupp 2011), may match the syntax or semantics of the ATM requirements. Of course, in cases where the pre-existing boilerplates do not match the requirements, they must be adapted in order to capture complex relationships between entities that are important to formulate independent of the ATM domain.

In general, when the requirements of the new field or new requirements do not match on the pre-existing ones then it is necessary to create new ones or adapt the pre-existing ones. It is observed that the basic assumption of the pre-existing boilerplate is the following: "*subject verb object*". Our methodology took advantage of the fact that the basic structure of pre-existing boilerplate is similar to the structure of the RDF triplet (*subject-predicate-object*). The complex boilerplate template consists of a small number of RDF triples (a small semantic graph). We have expanded with temporal and logical conditions the basic structure as well as with subordinate clauses and attributes accompanying the subject and object (Table 2). So, more complex requirements are implemented or modelled with complex boilerplates and by extension modelled as sets of connected RDF triples known as semantic graphs. The classification of our boilerplates is: (a) basic boilerplate template and extended form of basic boilerplate template (b) and boilerplate template with temporal or logical conditions. Also, Table 3 shows the classification.

Regarding the categories of boilerplates, the basic and simplest structure follows the RDF triples and has the following syntax <*subject*> <*verb*> <*object*>. Examples of the basic boilerplate and the extended basic boilerplate are shown in Figs. 5 and 6 respectively. Table 4 presents requirements in natural language and the respective requirements of basic and extended boilerplates (Table 5). Table 6 depicts the attributes of boilerplates and examples of values.

As we defined above, the basic template is as follows: <*subject*> <*verb*> <*object*>. An example of a requirement is: *ATM returns the CashCard* and the corresponding boilerplate is the basic boilerplate (<*subject*> <*verb*> <*object*>) which corresponds to the following boilerplate

Table 2 Comparison of EARS, Pohl and Rupp and our boilerplates

Boilerplates of Pohl and Rupp (Pohl and Rupp 2011)	EARS boilerplates (Mavin et al. 2009)	Our boilerplates
Basic: *<System name> shall/ should/will/may <process verb>*	*Generic requirements syntax:* *<optional preconditions> <optional trigger> the <system name> shall <system response>*	*Basic:* *<subject> <verb> <object>*
Complete process verb: *<System name> shall/ should/will/may <process verb> <object> <additional details about object>*	*Ubiquitous requirements:* *The <system name> shall <system response>*	*Extended basic boilerplate:* *<subject> <verb> <object> [of entity] + To <entity>* *<subject> <verb> <object> [of entity] + From <entity>*
Logical and temporal condition: *<When> <System name> shall/ should/will/may <process verb> <object> <additional details about object>*	*Event-driven requirements:* *WHEN <optional preconditions> <trigger> the <system name> shall <system response>*	*Logical and temporal condition:* *Basic logical condition boilerplate:* *if basic boilerplate + then basic boilerplate +* *Extended logical condition boilerplate:* *if basic boilerplate + then basic boilerplate + else* *basic boilerplate +* *Nested if condition boilerplate:* *if basic boilerplate + then* *if basic boilerplate + then* *basic boilerplate +* *Temporal condition boilerplate:* *After basic boilerplate, basic boilerplate* *When basic boilerplate, basic boilerplate*
	Unwanted behaviours: *IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>*	

Table 3 Categories of our boilerplates

Category	Boilerplate	Description
1. *Basic boilerplate*	*< subject > < verb > < object >*	This boilerplate is called basic boilerplate
1.1 *Extended basic boilerplate with details*	*< subject > < verb > < object >* *[details] < subject > < verb > < object >,* *< subject > [details] < verb > < object >,* *< subject > < verb > [details] < object >,* *< subject > < verb > < object > [details],* *[details] < subject > [details] < verb > [details] < object > [details]*	[Details] can also be applied anywhere. [Details] are explanatory comments
1.2 *Extended basic boilerplate with subjects as properties of classes*	*< subject > of < entity > < verb > < object >*	*subjects as properties of classes: < subject > of < entity > < verb > < object >*
1.3 *Extended basic boilerplate with objects as properties of classes*	*< subject > < verb > < object > of < entity >*	*objects as properties of classes: < subject > < verb > < object > of < entity >*
1.4 *Extended basic boilerplate with subjects as properties of classes and objects as properties of classes*	*< subject > of < entity > < verb > < object > of < entity >*	*objects as properties of classes, subjects as properties of classes* *< subject > of < entity > < verb > < object > of < entity >*
1,5 *Extended basic boilerplate with interactionand many objects*	*< subject > < verb > < object > + From < entity >* *< subject > < verb > < object > + To < entity*	*< subject > < verb > < object > + From < entity >* *< subject > < verb > < object > + To < entity*
1,6 *Extended basic boilerplate with interactionand objects as properties of classes*	*< subject > < verb > < object > of < entity > + To < entity >* *< subject > < verb > < object > of < entity > + From < entity >*	*< subject > < verb > < object > of < entity > + To < entity >* *< subject > < verb > < object > of < entity > + From < entity >* *Usually. the interaction is accompanied by objects as properties of classes*

Table 3 (continued)

Category	Boilerplate	Description
2 *Basic logical condition boilerplate*	*if basic boilerplate* + *then basic boilerplate* +	Extends the basic template with the if statement. This boilerplate is called conditional boilerplate. After *if* we can have more than one basic boilerplate which are separated by logical operators. Also, after *then* we can have more than one basic boilerplate

2.1
Extended logical condition boilerplate
	if basic boilerplate + *then basic boilerplate* +
	else
	basic boilerplate +
		if the condition is false, the else statement boilerplate will be executed

2.2 Nested if Condition boilerplate	*if basic boilerplate* + *then* *if basic boilerplate* + *then* *basic boilerplate* +	Nested if functions boilerplates
3 *Temporal condition* (*After or When*)	*After basic boilerplate, basic boilerplate* *When basic boilerplate, basic boilerplate*	Extends the basic template with temporal connectives to create sentences that involve temporal relations between entities. This boilerplate is called temporal boilerplate

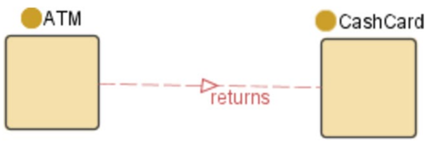


Fig. 5 Example of Basic Boilerplate

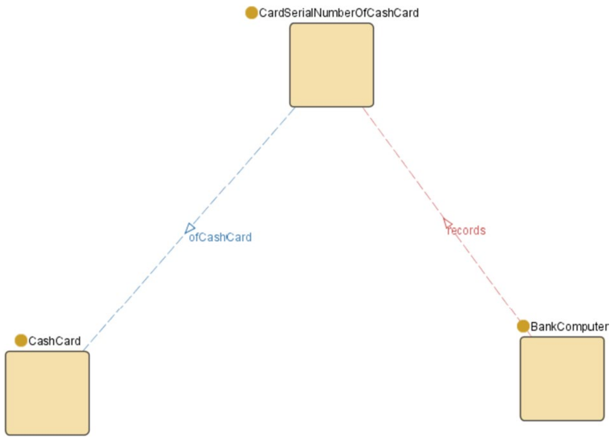


Fig. 6 Example of the extended template 1

instantiation: <*ATM*> <*returns*> <*CashCard*>. The verb (or predicate or object property in RDF syntax) is the *returns.* The domain of the object property in the ontology is class *ATM* which is the subject in the boilerplate. The range of the object property in the ontology is class *CashCard* which is the object in the boilerplate. In both cases, domain and range, we notice that ATM and Card are classes in our ontology.

However, there are cases where the object of the verb cannot be a class. For example, the following requirement: *The bank's computer records the serial number of the card*. The serial number in the ontology cannot be defined as a class but as a data property with domain *CashCard*, because it is a data property. For this purpose, it is useful to define a class *CardSerialNumberOfCashCard*. We also created a new object property which is called *ofCashCard.* This object property (*predicate*) has as domain the class *CardSerialNumberOfCashCard* and as range the class *CashCard.* The data property *cardSerialNumber* has as domain the class *CardSerialNumberOfCashCard*. The class *CardSerialNumberOfCashCard* we created is called metaclass. So, the RDF triple is represented in the ontology as *subject-predicate-object*, such as *BankComputer records CardSerialNumberOfCashCard,* because the above triple cannot have as object the data property *cardSerialNumber*. Table 5 shows an example of an extended boilerplate with elements from the ontology. We have followed the same rationale for the subject.

A large part of the requirements has verbs which have as objects data properties or have as subject data properties. That's why we expanded the basic structure

Table 4 Examples of basic and extended boilerplates

No	Requirements expressed in natural language	Requirements expressed using the Basic boilerplate	Category or template of basic boilerplate	Description
1	Return cash card	*<ATM> <returns> <CashCard>*	*<subject> <verb> <object>*	*Basic boilerplatte*
2	The serial number should be logged	*<BankComputer> <records> <cardSerialnumber> of<CashCard>*	*<subject> <verb> <object> of<entity>*	*Extended basic boilerplate objects as properties of classes*
3	Send serial number and password to bank computer	*<ATM> <sends> <typedpassword> of<Customer> <cardSerialNumber> of<CashCard> To<BankComputer>*	*<subject> <verb> <object> of<entity> +To<entity>*	*Extended basic boilerplate with interaction and objects as properties of classes*
4	The amount of cash is less than t	*<transactionAmount> of<ATMTransaction> <is_less_than_or_equals_to> <accountMaxWithdrawalPerDayAndAccount> of<Account>*	*<subject> of<entity> <verb> <object> of<entity>*	*Extended basic boilerplate subjects as properties of classes and objects as properties of classes*
5	Receive response from bank (about authorization)	*<ATM> <receives> <rejectionAutorization> From<BankComputer>*	*<subject> <verb> <object> of<entity> +From<entity>*	*Extended basic boilerplate with interaction*

Table 5 Example of an extended boilerplate with elements from the ontology

Requirement	*Computer records the cardserialNumber*
Category	*Extended basic boilerplate*
Boilerplate	*< subject > < verb > < object > of < entity >*
RDF triplet in ontology	*BankComputer records CardSerialNumberOfCashCard*
objectOfEntity (class)	*CardSerialNumberOfCashCard*
Object property	*ofCashCard*
Domain of object property (OfCashCard)	*CardSerialNumberOfCashCard*
Range of object property (OfCashCard)	*CashCard*
Domain of data property (cardSerialNumber)	*CardSerialNumberOfCashCard*

Table 6 Boilerplates attributes with values

Boilerplate attribute	Description	Example of values from ontology
< subject >	Class, data property (in case of subject of entity)	ATM, Customer, CashCard, Maintainer
< verb >	predicate	returns, displays, sends
< object >	instances, data property (in case of object of entity), class	CashCard, cardSerialNumber (data property), has_no_money (instances)
of < entity >	class	Customer, CashCard
To < entity >	class	ATM, BankComputer
From < entity >	class	ATM, BankComputer

boilerplate, in order to have the basic boilerplate with objects as properties of classes and subjects as properties of classes, such as the following:

< subject > < verb > < object > of < entity >

< subject > of < entity > < verb > < object >

< subject > of < entity > < verb > < object > of < entity >

Pohl and Rupp (2011) refer to the interaction between systems or interface requirement that is, a system performs an activity that depends on other systems. More specifically when a system receives a message and according to it, it must perform a function or perform the appropriate behavior. They suggest an appropriate template which is related to interface requirement:

The < system name > shall/should/will/may be able to < process verb > < object > .

According to the requirements of the ATM domain, we observed that the interaction between computer-system occupies a large part of the requirements such as: ATM sending serial number for checking to bank computer. The bank's processor sends an authorization to withdraw money. The ATM receives the authorization to withdraw and proceeds with the corresponding behavior or operation. In general, in the specific case study, we observe that there is an interaction between the ATM and the bank's computer. For this reason, we created the corresponding template for interaction. Specifically, we extended the basic boilerplate template with interaction and many objects and created the templates below:

$<subject> <verb> <object> of <entity> + To <entity>$

$<subject> <verb> <object> of <entity> + From <entity>$

$<subject> <verb> <object> + From <entity>$

$<subject> <verb> <object> + To <entity$

Figure 5 gives a simple example of an instantiation of the basic boilerplate $<subject> <verb> <object>$, namely $<ATM> <returns> <CashCard>$. The object property *returns* has domain the class *ATM* and range the class *CashCard.*

An example of the template $<subject> <verb> <object> of <entity>$ is $<BankComputer> <records> <cardSerialnumber> of <CashCard>$, which is depicted in Fig. 6. The *BankComputer* is a class, *records* is an object property. The *cardSerialnumber* is a datatype property. Object properties links classes. In this case, *cardSerialnumber* is a datatype property with domain the class *CardSerialNumberOfCashCard*. The object property *records* has domain the class *BankComputer* and range the class *CardSerialNumberOfCashCard.* Also, we have created the object property *ofCashCard*, which has domain the class *CardSerialNumberOfCashCard* and range the class *CashCard.*

An extended form of the basic boilerplate with interaction and object (data property) of entity is $<subject> <verb> <object> of <entity> + To <entity>$. This example is shown in Fig. 7. The following example $<ATM> <sends> < typedpassword > of < Customer > < cardSerialNumber > of < CashCard> To <BankComputer>$ is an instance of the above template. The object property *sends* has domain the class *ATM* and range the class *TypedPasswordOfCustomerCardSerialNumberOfCashCardToBankComputer.* *Typedpassword* is a data property with domain *TypedPasswordOfCustomerCardSerialNumberOfCashCardToBankComputer.* The *cardSerialnumber* is a

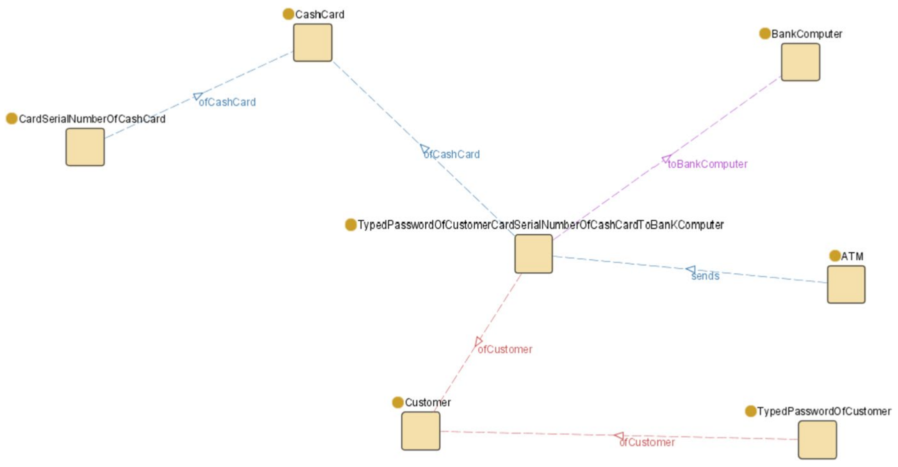


Fig. 7 Example of the extended template 2

datatype property with domain *Typed PasswordOfCustomer CardSerialNumberOfCashCardToBankComputer.* The object property *ofCashCard* has domain the class *TypedPasswordOfCustomerCardSerialNumberOfCashCardToBankComputer* and range the class *CashCard.* The object property *ofCustomer* has domain the class *TypedPasswordOfCustomerCardSerialNumberOfCashCardToBankComputer* and range the class *Customer.* The object property *sends* has domain the class *BankComputer* and range the class *TypedPasswordOfCustomerCardSerialNumberOfCashCardToBankComputer.*

Figure 8, below, shows the following boilerplate < *subject* > *of* < *entity* > < *verb* > < *object* > *of* < *entity* >. An example of this boilerplate is the < *transactionAmount* > *of* < *ATMTransaction* > < *is_less_than_or_equals_to* > < *accountMaxWithdrawalPerDayAndAccount* > *of* < *Account* >. The object property *is_less_than_or_equals_to* has domain the class *TransactionAmountOfTransaction* and range the class *AccountMaxWithdrawalPerDayAndAccountOfAccount.* The datatype property *accountMaxWithdrawalPerDayAndAccount* has domain the class *AccountMaxWithdrawalPerDayAndAccountOfAccount.* The *transactionAmount* is a datatype property with domain *TransactionAmountOfTransaction.* The object property *ofTransaction* has domain the class *TransactionAmountOfTransaction* and range the class *ATMTransaction.* The object property *ofAccount* has domain the class *AccountMaxWithdrawalPerDayAndAccountOfAccount* and range the class *Account.* Table 7 shows an example of an extended boilerplate with elements from the ontology.

Another example of an extended template of the basic boilerplate is the following*:* < *subject* > < *verbs* > < *object* > + *From* < *entity* >. Figure 9 describes its instantiation < *ATM* > < *receives* > < *rejectionAutorization* > *From* < *BankComputer* >. The object property *receives* has domain the class *ATM* and range the class *InteractionFromBankComputer.* The entity *rejectionAutorization* is an individual of class *Negative,* which is subclass of *Response.* The class *Response* is subclass of *Interaction*. The object property *fromBankComputer* has domain the class *InteractionFromBankComputer* and range the class *BankComputer.*

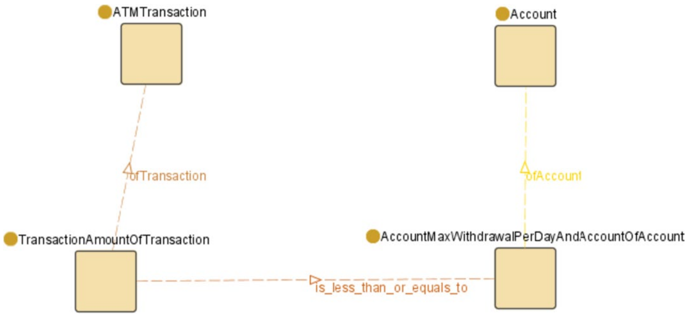

ATMTransaction
Account
ofTransaction
ofAccount
TransactionAmountOfTransaction
AccountMaxWithdrawalPerDayAndAccountOfAccount
is_less_than_or_equals_to

Fig. 8 Example of the extended template 3

Table 7 Example of an extended boilerplate with elements from the ontology

Category	*Extended BasicBoilerplate*
Boilerplate	*<subject> of <entity> <verb> <object> of <entity>*
Completed from ontology	*<transactionAmount> of <ATMTransaction> <is_less_than_or_equals_to> <accountMaxWithdrawalPerDay-AndAccount> of <Account>*
RDF triplet	*TrancactionAmoutofTransaction is_less_than_or_equals_to AccountMaxWithdrawalDayandAccountOfAccount*
Object of entity (class)	*AccountMaxWithdrawalDayandAccountOfAccount*
Object property	*ofAccount*
Domain of object property	*AccountMaxWithdrawalDayandAccountOfAccount*
Range of object property	*Account*
Data property	*accountMaxWithdrawalDayandAccount*
Domain of data property	*AccountMaxWithdrawalDayandAccountOfAccount*

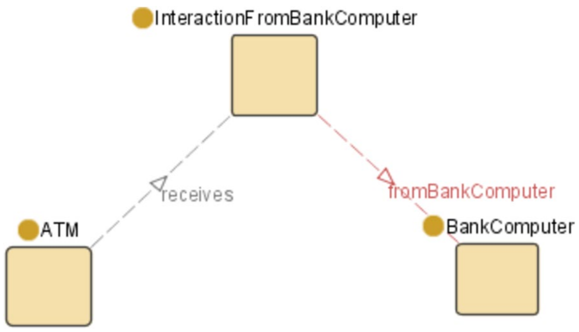


Fig. 9 Example of the extended template 4

4 A tool based on boilerplate and ontology for specifying requirements

In this work, we created a tool which is used to record the requirements using ontology and boilerplates. So far, in many systems that use boilerplates to document requirements and the attributes of boilerplates are completed by the ontology. In our case, whole boilerplates and requirements are stored in the ontology. The requirements engineer types the requirement in natural language and selects the appropriate boilerplate template. The tool accepts the requirement and performs parsing to detect the relation-verb of the sentence. The tool gives options (attributes) for its components of boilerplates. The user selects the appropriate resources to create boilerplates-based requirement. The purpose of the tool is to suggest possible values for the attributes of boilerplates. Figure 10 depicts the architecture of our tool.

First, it accepts the requirement in natural language and the parser searches for the verb of the requirement. The object property extractor accepts the result of the parser and looks for a corresponding or similar object property in the ontology. The algorithm is shown in Listing 1.

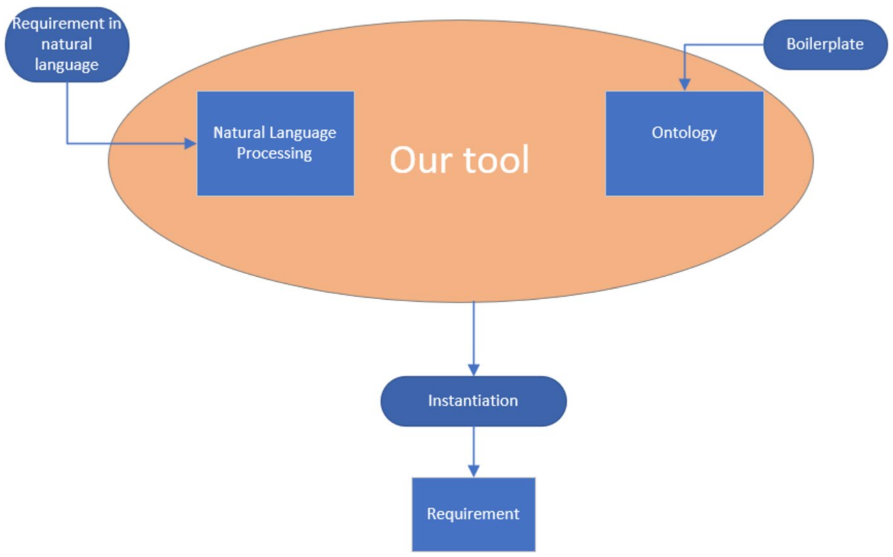


Fig. 10 Tool for constructing requirements based on ontology and boilerplates

```
objectPropertyExtractor(String verb){
    listOfObjectProperties=instatiate from ontology();
    String i="";
    String nameOfObjectProperty;
    while(listOfObjectProperties not null){
        i=currentObjectProperty. istOfObjectProperties();
        if (i.equals(verb){
            nameOfObjectProperty=verb;
        }
    }
   return listOfObjectProperties;
}
```

Listing 1 Algorithm of the object property extractor

The range extractor accepts the object property and returns the range of property. Depending on the boilerplate which is chosen by the user for the requirement, the method, range extractor, is different depending on the cases we mentioned in the previous paragraph. For the basic boilerplate <*subject*> <*verb*> <*object*>, such as <*ATM*> <*returns*> <*CashCard*> *or* <*ATM*> <*displays*> <*has_no_money*> whose <*object*>-part is an instance; for such cases, we have also created the corresponding module, namely the instances extractor. The algorithm is shown in Listing 2.

Also, one of the extended templates of the basic boilerplate is: <*subject*> <*verb*> <*object*> *of* <*entity*>. For example, the following requirement <*BankComputer*> <*records*> <*cardSerialNumber*> *of* <*CashCard*> is an

```
RangeExtractor(Object Property){
listRangeOfProperty=instatiate from ontology();
String nameRange="";
String range;
String instances;
listNameofRange;
     while(listRangeOfProperty not null){
         range =currentRange.listRangeOfProperty ();
         if (range is only one class){
                  nameRange =getName();
                 listNameofRange .add(nameRange);
                 if (nameRange has instances){
                     instances.getInstances();
                 }
         }
        else{
           while (range not null){
                 nameRange =getName();
                 listNameofRange .add(nameRange);
                  if (nameRange has instances){
                     instances.getInstances();
                  }
           }
        }
     retun listRangeOfProperty;
     }
```

Listing 2 Algorithm of the range extractor

instantiation of the above template. The object property is *records*. This object property has range *CardSerialNumberOfCashCard* class. Also, there is another object property, *ofCashCard* which has domain the *CardSerialNumberOfCashCard* and range the *CashCard* class. Additionally, for this kind of template we had to define as range of the datatype property *cardSerialNumber* the *CardSerialNumberOfCashCard* class. In this case, the object is not a class or an instance in the basic template, but it essentially has a metaclass as described above. So that's why we created the extractor *objectOfEntity*.

The following (extractor *objectOfEntity*) works with the assumption that the range extractor is implemented. To complete the relation, we use the range extractor and to complete the subject the domain extractor. There is the domain extractor, which accepts an object property and finds its domain. In the above case, it looks for an object property (e.g. *ofCashCard*) and a data property (e.g. *cardSerialNumber*) which have the same domain (e.g. *cardSerialNumberOfCashCard*) with the range (e.g. *cardSerialNumberOfCashCard*) of the relation, in this case *records*. After they are found, then we take the range (e.g. *CashCard*) of the object property to insert it in the position of the attribute *of<entity>*. Also, in the position *<object>*, we insert the data property.

A second example of the extended template is: *<subject> <verb> <object> of <entity> + To <entity>*. An instantiation for this template is as follows: *<ATM> <sends> <typedpassword> of <Cus-*

tomer > To < BankComputer >. It is similar to the above example. Besides the datatype property *typedpassword* and the object property *ofCustomer* it also has one more object property (*ToBankComputer* This object property has range *TypedPassowordOfCustomerToBankComputer* class. Also, there is another object property, *ofCustomer* which has domain the *TypedPassowordOfCustomerToBankComputer* and range the *Customer* class. Additionally, for this kind of template we had to define as range of the *typedpassword* datatype property the *TypedPassowordOfCustomerToBankComputer* class. A third example of the extended template is the following*: < subject > < verbs > < object > of < entity > From < entity >* which is similar to the previous one, but it has additional object property *fromSomething*. One example requirement that follows this boilerplate is the following: *< BankComputer > receives < transactionAmount > of < Transaction > From < ATM >*.

To complete the relation, we use the range extractor and to complete the subject the domain extractor. In the above case, it looks for two object properties (*ofCustomer*), (*ToBankComputer*) and a data property (*typedpassword*) which have the same domain (*typedPasswordofCustomerToBankComputer*) with the range of the relation, in this case *sends*. After they are found, then we take the range (e.g. *Customer*) of the object property to insert it in the position of the attribute *of < entity >* and we e.g. take the range (e.g. *BankComputer*) of the second object property (e.g. *ToBankComputer)* to insert it in the position of the attribute *To < entity >*. Also, in the position *< object >*, we insert the data property. To complete the *< object > of < entity > To < entity >* we use the extractor objectOfEntity. The algorithm for the above extended case is shown in Listing 3.

Note that the code inside "*else{}*" is similar to the one in "*if (ontclass and ontclass2 only one class){*", the only difference being that the domain of ontoclass2 is not just one class but many classes (e.g. in the case of union of classes). In this case an iterator must be used which each time takes the current class to check it. All modules are illustrated in Fig. 11.

A fourth example of the extended template is the following: *< subject > < verb > < object > of < entity > + From < entity > or subject > < sends > < object > of < entity > + To < entity >*. The difference with the previous ones is that the verb (object property) has more than one objects. For example, this boilerplate: *< BankComputer > receives < typedpassword > of < Customer > < cardSerialNumber > of < CashCard > From < ATM >* has two objects *(typedpassword* and *cardSerialNumber)*. Otherwise, it is similar to the above examples.

So far, we have seen that the datatype property *ofSomething* is in the object-part of the boilerplate. For this reason, we use extractor objectOfEntity, but it is also possible to be in the subject-part. The following are examples of such an extended boilerplate: *< subject > of < entity > < verb > < object > or < subject > of < entity > < verb > < object > of < entity >*. For example, the requirement *< transactionAmount > of < Transaction > < is_less_than_or_equals_to > < accountMaxWithdrawalPerDayAndAccount > of < Account >* has the *is_less_than_or_equals_to* as object property. This object property has domain *TransactionAmountOfTransaction* class. Also, there is another object property *ofTransaction*, which has domain the *TransactionAmountOfTransaction* class and

```
ExtractorObjectOfEntity( int found ){   //found=1 for one object property, 2 for two objects
properties.
    HashMap <String, ArrayList<String>> ObjectOfEntity = new HashMap <String,
ArrayList <String>>();
   RangeListFromEntity;
   String nameRange;
   String nameObject;
   String nameOfEntity;
   OntResource ontclass;
   OntResource ontclass2;
   OntResource ontclass3;
   while(listRange not null){
      nameRange=currentRange. listRange();
      listDataProperties=instatiate from ontology();
      while(listDataProperties not null){
         dataProperty=current.listDataProperties();
         listObjectProperties=instatiate from ontology();
         while(listObjectProperties not null){
         objectProperty=current. listObjectProperties();
         ontclass=dataProperty.getDomain();
         ontclass2=objectProperty.getDomain();
         if (ontclass and ontclass2 only one class){
            if (found==1){
               if (ontclass and ontclass2 has same domain){
                  nameObject=dataProperty;
                  nameOfEntity=objectProperty.getRange();
                  ObjectOfEntity.put(nameObject, new ArrayList<String>());
                  ObjectOfEntity.get(nameObject).add(nameOfEntity);
               }
            }
           else //(found==2){
            if (ontclass and onclass2 has same domain){
               listObjectProperties2=instatiate from ontology();
               while(list listObjectProperties2){
                  objectProperty2=current. listObjectProperties2();
                  ontclass3= objectProperty2.getDomain();
                  if(onclass2 and ontclass3 has same domain){
                      nameObject=dataProperty;
                      nameOfEntity=objectProperty.getRange();
                      ObjectOfEntity.put(nameObject, new ArrayList<String>());
                     ObjectOfEntity.get(nameObject).add(nameOfEntity);
                     nameToORFromEntity=objectProperty.getRange();
                    RangeListFromEntity.add(nameToORFromEntity);

                  }
               }
            }
         }

      }
        else {}
        }
      }
  }
    return ObjectOfEntity;
}
```

Listing 3 Algorithm of the extractor of Object of Entity

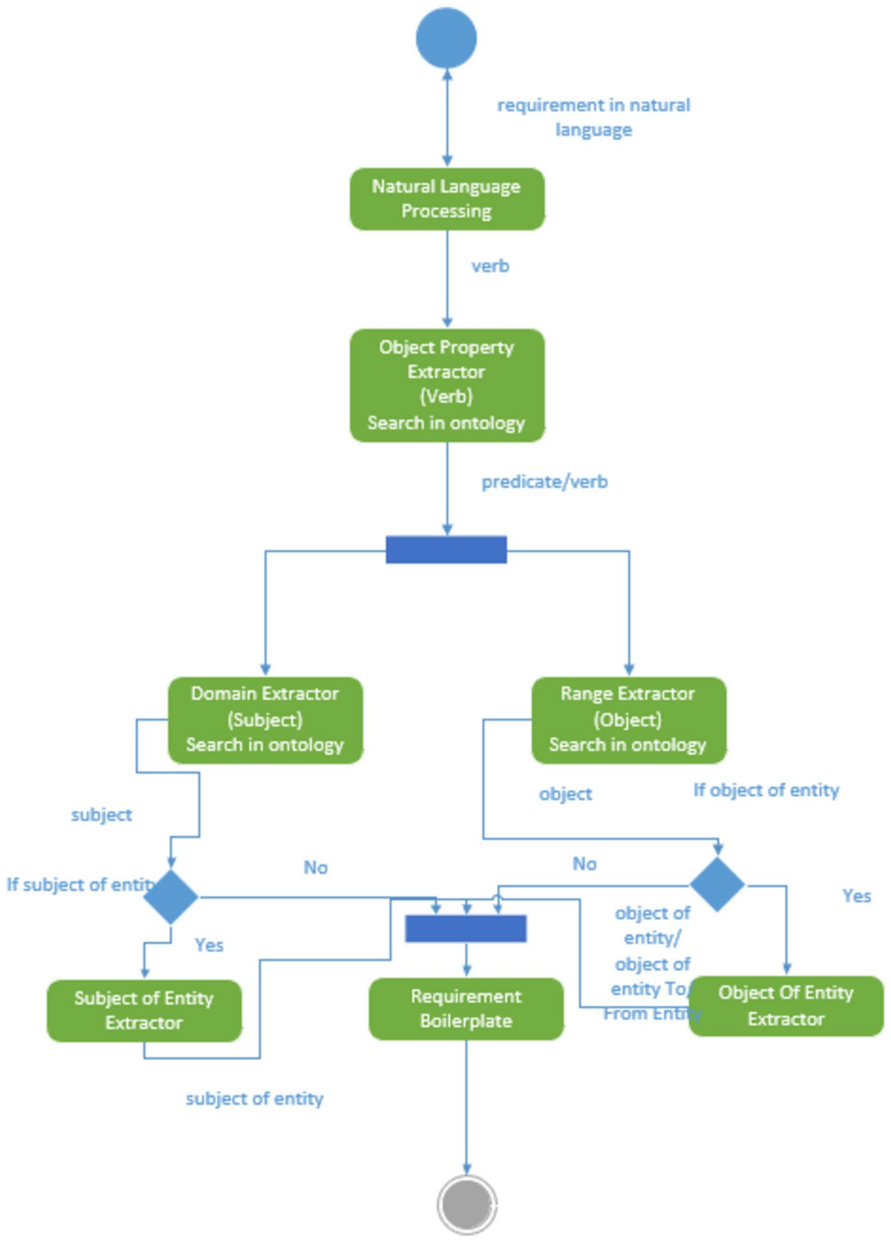

requirement in natural language
Natural Language Processing
verb
Object Property Extractor (Verb) Search in ontology
predicate/verb
Domain Extractor (Subject) Search in ontology
Range Extractor (Object) Search in ontology
subject
object
If object of entity
If subject of entity
No
No
Yes
Yes
object of entity/ object of entity To/ From Entity
Subject of Entity Extractor
Requirement Boilerplate
Object Of Entity Extractor
subject of entity

Fig. 11 Workflow of our tool

range the *Transaction* class. Additionally, the *transactionAmount* datatype property has domain the *TransactionAmountOfTransaction* class.

To complete the relation (*is_less_than_or_equals_to*), we use the range extractor. To complete the object of entity, it looks for an object property (e.g. *ofAccount*), and a data property (e.g. *accountMaxWithdrawalPerDayAndAccount*) which have the same domain (*accountMaxWithdrawalPerDayAndAccountOfAccount*) with the range of the relation, in this case *is_less_than_or_equals_to*. After they are found, then we take the range (*Account*) of the object property (e.g. *ofAccount*) to insert it in the position of the attribute (object) *of<entity>*. Also, in the position *<object>*, we insert the data property. To complete the *<object> of<entity>* we use the extractor *objectOfEntity*.

To complete the subject of entity, we look for an object property (e.g. *ofTransaction*), and a data property (e.g. *transactionAmount*) which have the same domain (*TransactionAmountOfTransaction*) with the domain of the relation, in this case *is_less_than_or_equals_to*. After they are found, then we take the range (*Transaction*) of the object property (e.g. *ofTransaction*) to insert it in the position of the attribute (subject) *of<entity>*. Also, in the position *<subject>*, we insert the data property (*transactionAmount*). To complete the *<subject> of<entity>*, we use the extractor *subjectOfEntity*. Therefore, apart from the extractor *objectOfEntity*, there is also the extractor subject of entity. Figure 11 presents the workflow of our tool.

5 Evaluation

We developed the tool in order to be useful for the requirement engineers in the process of specification of requirements. The tool is based on the methodology we proposed, namely we exploit the natural language syntax of boilerplates mapping them to RDF triples. This tool uses the domain-specific ontology as well as a minimal set of boilerplates which we developed. Regarding the use of the tool, we have to clarify that it does not require any learning about ontology technology but neither do engineers need to learn anything new to use this type of requirements, the boilerplates.

The functionality of the tool enables the user to add and edit a requirement based on the existing domain-specific ontology. The user types the requirement in natural language and selects the appropriate boilerplate template according to the language used in the requirement, so there is no need to know the boilerplate template in advance. The user selects the appropriate values of attributes coming from the ontology, from a list of options.

The user can choose the template of a basic boilerplate, or the extended template of a basic boilerplate or a complex boilerplate. The latter can contain logical and temporal constraints. In the case that a complex template is selected then the user needs to select from the first list the content of the complex template i.e., basic template or extended template. The processing of the tool for the choices of the attributes of boilerplates are based on the relation or the verb that the user types. The verb is derived from natural language processing. The tool detects relations with the use of Stanford Parser.[1]

After the user selects the appropriate boilerplate, options for each attribute are displayed in the window after being processed by the tool. Also, the user can

[1] https://stanfordnlp.github.io/CoreNLP/

choose between classes, instances, object properties and datatype properties. The user interface includes editing and adding requirements. The attributes (such as classes, instances, object properties and data properties) of boilerplates are based on a domain-specific ontology.

In order to assess the efficiency and effectiveness of the proposed methodology, an experiment was conducted on engineering software for an ATM. The experiment is an observational case study, and the design was carried out according to the template of Runeson et al.

6 Research questions

The experiment we are conducting aims to evaluate the effectiveness and efficiency of our method from the side of requirement engineers. This specific experiment is conducted in order to answer the following three research questions: (1) Are there discrepancies regarding the completion time of the engineers for a requirement specification using our method/tool? (2) How effective is the proposed methodology so that all participants have the same expectations for the tool?

Regarding the first research question, we measured the time that it takes for the participants to write a requirement in the system, to give the options and save the requirement instance in the system. The time to record the requirement is the answer to the question and the specific research question concerns the efficiency of our method. In the second research question, we assess the difficulty of choosing the appropriate boilerplate template and attribute values from the ontology given a description of the tool in natural language. The content of the research only concerns the evaluation of the proposed method which includes the tool used by the participants.

6.1 Description of the experiment

The developed tool is used to define requirements based on an ontology and boilerplates. Participants typed the requirement for the ATM system in natural language, selected the appropriate boilerplate template, the tool processed the users' choices and provided options of attributes boilerplates.

The three participants were given a brief description of the ATM ontology, the syntax of requirement boilerplates, examples of requirement boilerplates and the case study. Participants studied the above and were given access to our tool. They were not given any help while using the tool.

6.1.1 Data collection

After the participants completed defining the requirements with the tool given to them, qualitative data for research questions had to be collected. For the above reason, they were asked to answer a questionnaire in the form of a personal interview in order to collect qualitative data. The participants answered the Likert scale

Table 8 Questionnaire

Time for initial specification (in hours)
Overall understanding (0 not understanding -5 fully understood)
Boilerplate identification difficulty (0 not difficult – 5 very difficult)
Placeholder identification difficulty (0 not difficult – 5 very difficult)

questionnaire but also justified their answers. Table 8 presents the questionnaire. Finally, the evaluator asked for more information on the result of the requirements specification. The interviews were not recorded but the evaluator kept the data in the form of notes. The evaluator of the interviews is the same person who designed the tool, the syntax of the boilerplate and the ontology.

6.2 Data analysis

To draw conclusions about the experiment, the quantitative and qualitative data were analysed. Regarding the qualitative data, the values from the questionnaire among the participants were compared. In the quantitative analysis, the specification of the requirements that the participants were asked to implement through the tool was the subject of the evaluator's study. The evaluator counted the numbers of common boilerplates that were defined by the participants and the errors that occurred during the specification of the requirements.

6.3 Semantic analysis through SPARQL queries

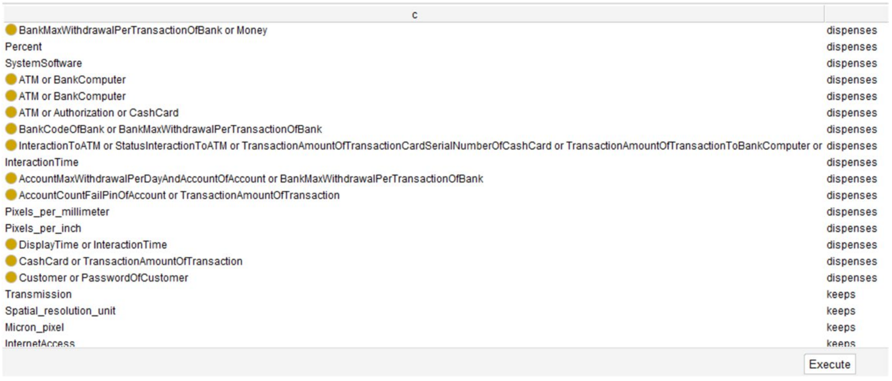
In order to ensure the validity and correctness of the requirements we have created some SPARQL queries that detect inconsistencies that would lead to semantic errors or detect possible requirements that have been omitted. SPARQL query is a query language for RDF data (graphs). In Listings 4 and 5, two such queries for checking requirements' incompleteness are presented. The first query looks for object properties whose domain is class ATM that do not have (yet) corresponding requirements. Indicatively, Fig. 12 shows the results from the first query. The second query looks

```
SELECT DISTINCT ?c ?p
  WHERE {
    ?p rdf:type/rdfs:subClassOf* owl:ObjectProperty .
    ?c rdf:type owl:Class .
    ?p rdfs:domain :ATM  .
    FILTER NOT EXISTS {
      ?r rdf:type/rdfs:subClassOf* :Requirement .
      ?r :verb ?p.
      ?r :subject :ATM .
    }
}
```

Listing 4 SPARQL query

```
SELECT DISTINCT  ?s ?p ?dp ?en
   WHERE {
      ?p rdf:type/rdfs:subClassOf* owl:ObjectProperty .
      ?o  rdf:type owl:Class .
      ?p  rdfs:range ?o .
      ?p  rdfs:domain :BankComputer .
      ?of  rdf:type/rdfs:subClassOf* owl:ObjectProperty .
      ?of  rdfs:domain ?o.
      ?en  rdf:type owl:Class .
      ?of  rdfs:range ?en.
      ?dp  rdf:type/rdfs:subClassOf* owl:DataProperty.
      ?dp rdfs:domain ?o .
      FILTER NOT EXISTS {
         ?r rdf:type/rdfs:subClassOf* :Requirement .
         ?r :subject :BankComputer.
         ?r :verb ?p.
         ?r :ofEntity  ?en  .
         ?r :object ?dp
      }
}
```

Listing 5 SPARQL query



c	
● BankMaxWithdrawalPerTransactionOfBank or Money	dispenses
Percent	dispenses
SystemSoftware	dispenses
● ATM or BankComputer	dispenses
● ATM or BankComputer	dispenses
● ATM or Authorization or CashCard	dispenses
● BankCodeOfBank or BankMaxWithdrawalPerTransactionOfBank	dispenses
● InteractionToATM or StatusInteractionToATM or TransactionAmountOfTransactionCardSerialNumberOfCashCard or TransactionAmountOfTransactionToBankComputer or	dispenses
InteractionTime	dispenses
● AccountMaxWithdrawalPerDayAndAccountOfAccount or BankMaxWithdrawalPerTransactionOfBank	dispenses
● AccountCountFailPinOfAccount or TransactionAmountOfTransaction	dispenses
Pixels_per_millimeter	dispenses
Pixels_per_inch	dispenses
● DisplayTime or InteractionTime	dispenses
● CashCard or TransactionAmountOfTransaction	dispenses
● Customer or PasswordOfCustomer	dispenses
Transmission	keeps
Spatial_resolution_unit	keeps
Micron_pixel	keeps
InternetAccess	keeps

Fig.12 Answers of first query

for object properties whose domain is class BankComputer and as object has meta-class (class) as in Fig. 6. Thus, the requirement engineer should continue to provide requirements for these properties returned by both queries (Table 9).

7 Results

The quantitative results, regarding the difficulty of identifying the correct boilerplate type, are recorded in Table 10. These results come from specifying the requirements implemented by the participants. Also, Table 10 contains the remaining quantitative

Table 9 Time in hours for requirements specification using our method

Time for initial specification (in hours)	Engineer1	Engineer2	Engineer3
<subject> <verb> <object>	00:19:71	00:48:88	00:20:43
<subject> <verb> <object> of<entity>	00:28:85	00:25:05	00:25:32
<subject> of<entity> <verb> <object> of<entity>	00:41:72	00:45:05	00:42:49
<subject> <receives> <object> of<entity> +From<entity>	01:15:19	00:56:39	00:54:12
Complex boilerplate	01:48:43	02:06:92	03:09:25

Table 10 Quantitative results

Question	Engineer1	Engineer2	Engineer3
Overall understanding	0	1	1
Boilerplate identification difficulty	1	1	1
Placeholder identification difficulty	0	0	1

results which are related to the overall understanding of the system and the difficulty of choosing values for attributes.

Regarding the proposed method as well as its tool, the participants did not encounter any particular difficulty and found them understandable and easy. Another reason that enhances the ease of use of the tool is that the ontology describes the ATM software, and by extension the boilerplate language is embedded in ATM concepts and relationships, so it was expected that basic knowledge of the specific software, ATM would facilitate its use. It is worth noting that once the ontology description was given, subjects showed familiarity with the concepts and relationships. This resulted in participants specifying requirements without the support of the grammar. Generally, in terms of the boilerplate language, participants reported that it is quite intelligible and expressive. Also, the participants mentioned during interviews that they do not need to learn anything new to understand and use it. They had no difficulty in choosing the correct boilerplate template nor they encountered any difficulty in choosing values for the attributes. Also, the participants have little experience in specifying requirements as well as little experience in the ATM domain. Nevertheless, they had no difficulty in choosing the right instances. It was also found during time recording that the participants needed more time on complex requirements as shown in Table 9. Finding the right instances is a time-consuming task and they also made mistakes until they found the right instances. Also, regarding the validity of the evaluation, I mention the following cases such as the very small sample and the homogeneity of the sample.

The first research question relates to whether there are differences in the completion time of requirements between engineers using our method. From the quantitative data, we observe that there are no large discrepancies between the participants. Nevertheless, there are differences in completion time between basic requirements and complex requirements.

The second research question refers to the evaluation of the effectiveness of the proposed methodology by the users. The participants did not face difficulties in understanding and using the methodology, nor difficulty in choosing boilerplate template, nor in choosing values for attributes.

8 Conclusions

We developed an tool that allows the requirements engineer to define requirements using natural language processing, a domain ontology and boilerplates. We have presented the proposed methodology that is embedded in the tool based on the ontology and the corresponding boilerplate language. We took advantage of the linguistic structure of RDF and have discussed the boilerplate language syntax and the classification of boilerplates. The most famous boilerplates templates are EARS and those of Pohl and Rupp from which we were inspired the categories of our boilerplate templates. In this paper, we focus on the following categories of boilerplates: (a) basic boilerplate template and extended form of basic boilerplate template and (b) and boilerplate template with temporal or logical conditions. We point out that the boilerplate syntax is similar to the linguistic structure of RDF. The above is what makes our methodology unique. For the development of the ontology and by extension the creation of boilerplates the ATM domain was used.

To avoid errors in subsequent stages of verification and validation of requirements, the creation of high-level requirements of a system are important. Natural language is still considered the most popular technique for specification requirements. Alas, natural language is ambiguous. Boilerplates deal with the ambiguity. Also, requirement engineers do not need special training to use boilerplates. So far, for the specification of requirements, we have seen the use of ontologies and boilerplates in combination. In several cases, the completion of the attributes of a boilerplate is done by entities from the ontology. The variants of Hull and EARS boilerplates have been mostly used.

Using our tool, the user or the requirement engineer types a requirement using natural language and selects the appropriate boilerplate template according to the linguistic structure of the requirement. The tool proposes options for the values of attributes. This is achieved by processing the requirement in natural language. Based on this process, the tool provides to the user matching boilerplate templates and then the user selects one of them and fill the templates placeholders with values from the ontology. The syntax of boilerplates is *subject – verb—object*. Similarly, the RDF triples also follow the same syntax *subject—predicate (verb phrase)—object*. The contribution and novelty of this work is that we exploit the syntax of the boilerplate language, mapping them to RDF triples. Therefore, not only the attributes of the boilerplates are completed by the ontology but the whole boilerplate is part of the ontology.

It is worth noting at this point that boilerplates have some fixed word and some attributes which are completed manually from the engineer or with help of the ontology. In this work, the tool, which we developed, based on the boilerplate language and the ontology, suggests options for attributes with the help of the ontology and via natural language processing. We exploit the syntax of the boilerplate language, mapping them to RDF triples. Mostly in other works, where they incorporate the combination of boilerplates and ontology, the attributes of boilerplates are completed with the help of ontology. This paper is that the whole boilerplate is stored in the ontology and attributes and fixed elements are part of the ontology. This constitutes the contribution and novelty of this work.

In order to ensure the quality and correctness of requirements, the combination of ontology and boilerplates is a useful tool for requirements engineers during requirements definition. Also, for non-experienced requirements engineers it is a good tool and guide them to formulate requirements. This combination has other advantages as well, such as reducing the ambiguity caused by natural language and the editing requirement specification. The reuse and renewal of the ontology and boilerplates are characterized as an advantage of the combination.

The proposed tool was evaluated for effectiveness and efficiency through the experiment conducted. The users who participated in the experiment had no difficulty in choosing the appropriate template boilerplate or choosing the attribute values. Also, they have no difficulty in understanding and using the proposed methodology. As future work, we will evaluate this tool using more participants and in the context of real-word software production.

Author contributions All authors wrote the manuscript.

Funding Open access funding provided by HEAL-Link Greece.

Declarations

Competing interests The authors declare no competing interests.

References

Ahmad, S., Anuar, U., Emran, N.A.: A tool-based boilerplate technique to improve SRS quality: an evaluation. J. Telecommun. Electron. Comput. Eng. (JTEC) **10**(27), 111–114 (2018)

Antoniou, G., Groth, P., Van Harmelen, F., Hoekstra, H.: A Semantic Web Primer (3rd edn.) MIT press (2011)

Anuar, U., Ahmad, S., Emran, N.A.: A simplified systematic literature review: improving software requirements specification quality with boilerplates. In: 2015 9th Malaysian Software Engineering Conference (MySEC), 99–105. IEEE. (2015) https://doi.org/10.1109/MySEC.2015.7475203

Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F.: Automated checking of conformance to requirements templates using natural language processing. IEEE Trans. Softw. Eng. **41**(10), 944–968 (2015). https://doi.org/10.1109/TSE.2015.2428709

Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F., Gnaga, R.: Automatic checking of conformance to requirement boilerplates via text chunking: an industrial case study. In: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, (pp. 35–44). IEEE. (2013) https://doi.org/10.1109/ESEM.2013.13

Arora, C., Sabetzadeh, M., Briand, L. C., Zimmer, F.: Requirement boilerplates: transition from manually enforced to automatically verifiable natural language patterns. In: 2014 IEEE 4th International Workshop on Requirements Patterns (RePa), (pp. 1–8) IEEE. (2014). https://doi.org/10.1109/RePa.2014.6894837

Daramola, O., Stålhane, T., Sindre, G., Omoronyia, I.: Enabling hazard identification from requirements and reuse-oriented HAZOP analysis. In: 2011 4th International Workshop on Managing Requirements Knowledge (pp. 3–11). IEEE. (2011) https://doi.org/10.1109/MARK.2011.6046555

Daramola, O., Sindre, G., Moser, T.: Ontology-based support for security requirements specification process. In: OTM Confederated International Conferences On the Move to Meaningful Internet Systems. (pp. 194–206). Springer, Berlin, Heidelberg. (2012) https://doi.org/10.1007/978-3-642-33618-8_28

Daramola, O., Sindre, G., Stalhane, T.: Pattern-based security requirements specification using ontologies and boilerplates. In: 2012 Second IEEE International Workshop on Requirements Patterns (RePa), (pp. 54–59). (2012) https://doi.org/10.1109/RePa.2012.6359973

Do, Q.A., Bhowmik, T., Bradshaw, G.L.: Capturing creative requirements via requirements reuse: a machine learning-based approach. J. Syst. Softw. **170**, 110730 (2020). https://doi.org/10.1016/j.jss.2020.110730

Do, Q. A., Chekuri, S. R., & Bhowmik, T.: Automated support to capture 1 creative requirements via requirements reuse. In: International Conference on 2 Software and Systems Reuse, (pp. 47–63). Springer, Cham. (2019) https://doi.org/10.1007/978-3-030-22888-0_4

Fanmuy, G., Fraga, A., Llorens, J.: Requirements verification in the industry. In: Complex Systems Design & Management (pp. 145–160). Springer, Berlin, Heidelberg. (2012). https://doi.org/10.1007/978-3-642-25203-7_10

Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Omoronyia, I., Zojer, H.: Ontology-driven guidance for requirements elicitation. In: Extended Semantic Web Conference (pp. 212–226). Springer, Berlin, Heidelberg. (2011) https://doi.org/10.1007/978-3-642-21064-8_15

Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., Panis, C.: DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In: 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (pp. 271–274). IEEE. (2011). https://doi.org/10.1109/DDECS.2011.5783092

Fritz, S., Srikanthan, V., Arbai, R., Sun, C., Ovtcharova, J., Wicaksono, H.: Automatic information extraction from text-based requirements. Int. J. Knowl. Eng. **7**(1), 8–13 (2021)

Gruber, T.R.: A translation approach to portable ontology specifications. Knowl. Acquis. **5**(2), 199–220 (1993). https://doi.org/10.1006/knac.1993.1008

Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. Int. J. Hum. Comput. Stud. **43**(5–6), 907–928 (1995). https://doi.org/10.1006/ijhc.1995.1081

Guarino, N., Oberle, D., Staab, S.: What is an ontology?. Handbook on ontologies, (pp. 1–17). (2009)

Haris, M.S., Kurniawan, T.A.: Automated requirement sentences extraction from software requirement specification document. In: Proceedings of the 5th International Conference on Sustainable Information Engineering and Technology (pp. 142–147), (2020)

Hull, E., Jackson, K., Dick, J.: Requirements engineering. Springer Science & Business Media, (2010)

Ibrahim, N., Kadir, W.M.W., Deris, S.: Propagating requirement change into software high level designs towards resilient software evolution. In: 2009 16th Asia-Pacific Software Engineering Conference (pp. 347–354). IEEE. (2009) https://doi.org/10.1109/APSEC.2009.55

Kravari, K., Antoniou, C., Bassiliades, N.: SENSE: a flow-down semantics-based requirements engineering framework. Algorithms **14**(10), 298 (2021). https://doi.org/10.3390/a14100298

Kravari, K., Antoniou, C., Bassiliades, N.: Towards a requirements engineering framework based on semantics. In: 24th Pan-Hellenic Conference on Informatics (pp. 72–76). (2020) https://doi.org/10.1145/3437120.3437278

Mahmud, N., Seceleanu, C., Ljungkrantz, O.: ReSA: an ontology-based requirement specification language tailored to automotive systems. In: 10th IEEE International Symposium on Industrial Embedded Systems (SIES), (pp. 1–10). IEEE. (2015) https://doi.org/10.1109/SIES.2015.7185035

Mahmud, N., Seceleanu, C., Ljungkrantz, O.: ReSA tool: structured requirements specification and SAT-based consistency-checking. In: 2016 Federated Conference on Computer Science and Information Systems (FedCSIS), (pp. 1737–1746). IEEE. (2016) https://doi.org/10.15439/2016F404

Mavin, A., Wilkinson, P., Harwood, A., Novak, M.: Easy approach to requirements syntax (EARS). In: 2009 17th IEEE International Requirements Engineering Conference, 317–322. IEEE. (2009) https://doi.org/10.1109/RE.2009.9

Mokos, K., Katsaros, P.: A survey on the formalisation of system requirements and their validation. Array **7**, 100030 (2020). https://doi.org/10.1016/j.array.2020.100030

Musen, M.A.: The protégé project: a look back and a look forward. AI Matters **1**(4), 4–12 (2015)

Pasquariello, A., Vitolo, F., Patalano, S.: Systems and requirements engineering: an approach and a software tool for the interactive and consistent functional requirement specification. In: International Joint Conference on Mechanics, Design Engineering & Advanced Manufacturing (pp. 491–502). Springer, Cham (2022)

Pohl, K., Rupp, C.: Requirements engineering fundamentals, (1st edn.), Rocky Nook Richard Cyganiak, David Wood, Markus Lanthaler, (eds.), RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation (2011) http://www.w3.org/TR/rdf11-concepts/

Runeson, P., Host, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples, 1st edn. Wiley, Hoboken (2012)

Staab, S., Studer, R. (eds.): Handbook on Ontologies (2nd edn). Springer. (2009) https://doi.org/10.1007/978-3-540-92673-3_0

Too, C.W., Hoo, M.H., Yen, W.W., Khor, K.C.: UReST: A knowledge-based usability requirements specification support tool. In: 2022 International Conference on Decision Aid Sciences and Applications (DASA) (pp. 1455–1459). IEEE. (2022)

Warnier, M., Condamines, A.: Improving requirement boilerplates using sequential pattern mining. In: Europhras Conference, November 2017, London. (2017). https://doi.org/10.26615/978-2-9701095-2-5_013

Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., Ibrahim, A.: Rcm: requirement capturing model for automated requirements formalisation. (2020) https://doi.org/10.48550/arXiv.2009.14683

Zichler, K., Helke, S.: R2BC: Tool-Based Requirements Preparation for Delta Analyses by Conversion into Boilerplates. In: Software Engineering (Workshops), ASE 2019: 16th Workshop on Automotive Software Engineering @ SE19, Stuttgart, Germany, pp. 45–52. (2019) http://ceur-ws.org/Vol-2308/ase2019paper03.pdf