Semantic Requirements Construction using Ontologies and Boilerplates

Christina Antoniou, Kalliopi Kravari, Nick Bassiliades
Department of Informatics, Aristotle University of Thessaloniki, Greece
{antoniouc,kkravari,nbassili}@csd.auth.gr

Abstract

This paper presents a combination of an ontology and boilerplates, which are requirements templates for the syntactic structure of individual requirements that try to alleviate the problem of ambiguity caused using natural language and make it easier for inexperienced engineers to create requirements. However, still the use of boilerplates restricts the use of natural language only syntactically and not semantically. Boilerplates consists of fixed and attributes elements. Using ontologies, restricts the vocabulary of the words used in the requirements boilerplates to entities, their properties and entity relationships that are semantically meaningful to the application domain, leading thus to fewer errors. In this work we combine the advantages of boilerplates and ontologies. Usually, the attributes of boilerplates are completed with the help of the ontology. The contribution of this paper is that the whole boilerplates are stored in the ontology, based on the fact that RDF triples have similar syntax to the boilerplate syntax, so that attributes and fixed elements are part of the ontology. This combination helps to construct semantically and syntactically correct requirements. The contribution and novelty of our method is that we exploit the natural language syntax of boilerplates mapping them to Resource Description Framework triples which have also a linguistic nature. In this paper we created and present the development of a domain-specific ontology as well as a minimal set of boilerplates for a specific application domain, namely that of engineering software for an ATM, while maintaining flexibility on the one hand and generality on the other.

Keywords: requirement boilerplates, requirements, ontology, requirement specification

1. Introduction

1.1 Requirements

Requirements are the cornerstone of system development, and the quality of requirements determines or influences the course (procedure) and progress of the project and consequently the success of the project. The documentation technique is used for communication between stakeholders; this can be natural language through the prose, more structured natural language-based text, or formal techniques. The documentation technique increases the quality of the documented requirements. The most well-known way to record or document the requirements of a system is the natural language and specifically the prose. The advantage of this type is that the stakeholder does not need to learn anything new to use this type and can be used for different kinds of requirements.

The advantage of natural language is comprehensibility and expressiveness. However, ambiguity and the description/formulation of complex expressions are considered as disadvantages of natural language [38].

Generally, in practice, natural language is used to document the requirements; however, there is an increasing use of models for documenting. These can be used in conjunction with natural language or replace the use of natural language. The most used models are goals models, use cases diagrams and conceptual models. Specific modeling languages are used to create conceptual models and each modeling language is defined by syntax and semantics. Conceptual modeling languages are divided into formal, informal, and semiformal depending on the degree of formalization. This type of documentation cannot be used universally as with the natural language used for all kinds of requirements. Also due to the high degree of formality, they face the difficulties and obstacles created by the use of natural language [38].

Regarding the definition of formal specification given by [29], it "… is the expression in some formal language and at some level of abstraction, of a collection of properties some system should satisfy". The formal specification includes the following components-features; first, it has a syntax, i.e., there are some rules to determine-define the sentences; second, it has semantics, meaning that there are again rules for interpreting the sentences accurately through a domain; and third, there are rules for extracting necessary-useful information from the specifications. The specification is essentially organized into units and the relationships between them. Each unit consists of the declaration part and an assertion part. Wleringa & Dubois [46] converge on the definition of formal specification using the conceptual modeling language of Pohl and Rupp [38], which does not include the third criterion. Also, formal specification helps to verify the requirements [7].

Furthermore, the formal specification differs from the semi-formal specification as the latter does not standardize the assertion part. Examples of semi-formal specifications are dataflow diagrams, entity relationship diagrams or state transition diagrams. In [8], [17], [18], [20], the use of conceptual models is characterized as a formal specification. Boilerplates are also considered semi-formal specification [2]. Do et al., [13] and Post et al., [39] consider natural language as an informal specification.

1.2 Boilerplates in Requirements

Intrinsically, natural language involves ambiguity and, in many cases, there are statements which can be interpreted with multiple interpretations. The identification and reading of requirements are carried out by people who have different knowledge, social background and experiences. Therefore, as there is an inhomogeneity in the above factors, people can interpret the information and by extension the requirements differently. When natural language is used to document requirements, it can lead to misunderstanding. So, in order to reduce the effects of natural language (language effects) the required templates are used. The definition given for requirements templates is: "a requirement template is a blueprint for the syntactic structure of individual requirements.". It is a good practice to use glossaries with the required templates, so that

stakeholders involved in the development process use the same terms. Therefore, a glossary contains definitions/terms [38]. These requirements templates are the so-called boilerplates. Most popular boilerplates are EARS [31] and Pohl and Rupp [38].

Hull et al., [24] state that boilerplates are a good way of standardizing the language through which the requirements are expressed. According to them, the process required to express a boilerplate requirement is to select the appropriate boilerplate from a pallet or collection and fill in the blanks with the appropriate data. For example, you look for the appropriate boilerplate for a requirement from the palette. Suppose that the following boilerplate is considered for a specific requirement: "The <system> shall <function> <object> every <performance> <units>". Then this becomes the following requirement: "The <coffee machine> shall <produce> <a hot drink> every <10> <seconds>". Therefore, there are some fixed elements and some elements that the user fills in. It is possible for many requirements to use the same boilerplate.

According to [24] the advantages of boilerplates are the following: a) they can be used to hide information or other confidential data such as in the military or commercial projects (project), and b) system information is easier to process, when the requirements are changed, the boilerplates corresponding to them is easily changed. When you try to match a requirement with a boilerplate and you find that there is not any suitable one, then a new boilerplate must be created. Finally, the use of boilerplates favors the existence of subtypes in the requirements. For example, the following boilerplate: "the communications system shall sustain telephone contact." can be created using the following subclass: The communications system shall sustain telephone contact with not less than 10 callers.". The above are mentioned by [24], but they do not consider them as advantages as we do.

Also, another advantage is that the number of boilerplates may be small, but in this small number great flexibility is maintained [16]. Furthermore, boilerplates help the inexperienced engineer to create the requirements [10], [11]. Also, the use of boilerplates contributes to the creation of high-quality requirements [16]. Problems related to the use of natural language in requirement specification, such as ambiguity, are reduced [33]. Another advantage is that boilerplates are reusable [25]. Warnier & Condamines [45] report that boilerplates are simple to use, easy to understand and do not require special training. Also, boilerplates can be combined to create new, more complex ones. It is also possible to create a boilerplates repository which can be renewed and will be a good base for the inexperienced requirements engineer [10]. Zichler & Helke [48] report that requirements models can be created through the boilerplates that allow requirements verification. Also, Zaki-Ismail et al., [47] mentions that boilerplates can be used for formalization.

Fanmuy et al. [14] mention that the combination of boilerplates with an ontology helps to create well-written requirements from the beginning (essentially saves time by verifying and confirming the requirements later). Also, the machine is able to better understand the meaning of a requirement, so the machine analyzes and stores conceptual information using this combination. The storage of inference rules together with the application of

artificial intelligence algorithms contributes to human-like reasoning, using ontologies and boilerplates for the implementation of the above. Finally, this combination facilitates detecting missing requirements, ambiguous requirements, noise in requirements and inconsistencies, in order to improve the requirements and reuse knowledge. Daramola et al., [10] and Daramola et al., [11] report that the combination of boilerplates and ontology reduces the effort for the Software Requirements Specification (SRS) process. It is also an auxiliary tool when there is no experience in requirements engineering or analysis, and finally, contributes to the quality of the requirements. The use of boilerplates with the help of an ontology gives additional advantages as it helps the analyst to fill the gaps of the boilerplate with the help of ontology also helps to discover the relationships between the requirements [11].

So far, we have seen the use of the combination of ontologies and boilerplates for the specification of requirements. Regarding the creation of boilerplates, the following works of [9], [10], [11] were based on the boilerplates of Hull [24], tailored, of course, to the needs of the field. Concerning the use of the ontology along with boilerplates in the above works we observe that attributes of boilerplates are registered or completed by the ontology, taking into account the ones mentioned above. Also, in several cases the combination of boilerplates with the ontology are used for requirement specification. This paper presents a combination of an ontology and boilerplates based on the linguistic nature of Resource Description Framework (RDF) triples, which is similar to the boilerplate syntax, in order to bridge the gap created by the ambiguity of natural language. RDF is a data model, which describes resources and semantic relations between them in the form of triples, namely statements consisting of a subject, a predicate and an object. Boilerplates also describe relationships; a verb is always used, framed by a subject and an object. In other words, a basic form of a boilerplate has a similar syntax to an RDF triple. The goal of our research is to eliminate the semantical ambiguity of natural language with the help of boilerplates. In this work, the basic boilerplate is represented as an RDF triple. Similarly, more complex boilerplates are represented by a small set of RDF triples, namely a small semantic graph. Usually, the various parts of a boilerplate are filled by the engineer manually, but in this work, we propose that this should be restricted to classes and properties of an ontology. The contribution and novelty of this work is that we exploit the natural language syntax of boilerplates mapping them to RDF triples which have also a linguistic nature.

The most common way to document requirements is in natural language. The advantage of natural language is that the stakeholder does not need to learn anything new to document the requirements in natural language. Also, natural language can be used in any kind of requirement. On the other hand, the disadvantage of natural language is ambiguity. One of the solutions for specifying requirements in order to deal with the problem of ambiguity is boilerplates. Using boilerplates, a kind of syntactically pre-defined natural language requirement patterns, is not difficult for use and the requirements engineer does not need to learn anything new to be able to use them. Boilerplates are used similar to the natural language but with a more limited syntax. The use of boilerplates limits the problems created by natural language such as ambiguity

Pohl and Rupp [37]. Boilerplates consist of fixed elements and attribute elements. The requirement engineer completes the attribute elements to construct the relevant requirements. Until now, the gaps of attributes elements are filled manually or from the ontology. The research question is the semantic and syntactic improvement of boilerplates by storing whole boilerplates (attributes and fixed elements) in the ontology and not only the items to be completed.

The rest of the paper is organized as follows. In section 2 we first briefly present related work. In section 3, we present the two most famous boilerplate schemes in the literature, namely EARS and the one of Pohl and Rupp, which our own methodology extends. In section 4 we present the ontology for the ATM domain. In section 5 we present and discuss the creation and classification of boilerplates based on the linguistic nature of RDF. We were also inspired this classification by the most famous boilerplates which are EARS and the one of Pohl and Rupp. The classification we created is based on a) requirements templates without conditions, b) requirements template with conditions, which are distinguished into logical and temporal conditionals, and c) boilerplates based on the linguistic nature of RDF triples that is similar to the syntax of the boilerplates. Finally, we present the conclusions in section 6.

2. Related Work

Boilerplates are used to reduce the ambiguity issue caused by the use of natural language. Below the various uses of boilerplates in the literature are presented. In subsection 2.1, we refer to the combination of boilerplates with NLP, in section 2.2 we are reviewing on the combination of boilerplates with ontologies, and in section 2.3 we present literature about the triple combination of boilerplates with ontologies and NLP.

Mavin et al., [31] created the EARS boilerplates which enable the requirements written in natural language to be represented in these templates. These templates were applied to the requirements of an aero engine control system by an airworthiness regulation document. The requirements templates were compared with the requirements in the natural language and the results showed that boilerplates are considered an effective tool for writing stakeholders' requirements.

Mavin & Wilkinson [32] customize the EARS templates to meet the case studies. The results of their paper converge with the above, i.e., EARS templates are an effective tool for writing high quality requirements.

The research of [2] conducted on the issue of how boilerplates improve the quality of SRS and concluded that the contribution of boilerplates to quality improvement in the process of SRS. Boilerplate is a useful tool to deal with the multidimensional phenomenon of quality (such as disambiguation, conformity, completeness, accuracy, reusability, etc.). The biggest help of boilerplates is in the characteristic of quality called ambiguity as it is the one that gathered the largest percentage from other characteristics.

Campanile et al., [9], state that one approach to deal with the ambiguity problem and formulate effective requirements is Behavioural-Driven Development (BDD). They focus

on the metric requirements in BBD. For the above issue they use traditional Natural Language Processing (NLP) metrics and a sample of requirements that has been rewritten with BBD criteria. Their paper highlights the importance of new metrics for BDD requirements specifications.

The work of Rosadini et al., [40] tackles with the detection of defects in requirements of railway signaling manufacturer domain based on NLP techniques. They identified defect categories and for each category a set of defect-detection patterns using the GATE tool. These patterns were applied to requirements which had defects. the comparison of traditional and NLP techniques ended with discrepancies. The analysis of discrepancies revealed useful hints to improve the NLP techniques.

Another work of Warnier & Condamines [45] in order to bridge the gap between requirements and requirements engineers and consequently to reduce the ambiguity in the specifications follow the solution of boilerplates. According to them, they consider that the boilerplates that exist in the literature are very general, yet they should be applicable in various fields. Therefore, they report that the role of boilerplates is partially fulfilled as they leave engineers a great deal of freedom to fill in the gaps. They propose a bottom-up approach, based on mining frequent textual patterns that occur in corpuses of genuine requirements written in natural language, in order to discover elements that could be used as boilerplates or elements that could create boilerplates.

Flemström et al., [19] refer to test cases which take place when the system is checked in order to detect errors due to unpredictable interactions. However, the disadvantage of these tests cases lies in the difficulty of expression. One reason is that they are based on complex mathematical notations. They present a solution to this problem based on boilerplates and specifically on the T-EARS (Timed Easy Approach to Requirements Syntax) language prototype.

2.1Detailed Background on Boilerplates

In this section, we describe how we create appropriate boilerplates. Boilerplates must be few in number and general, in order to maintain the characteristic of flexibility but also to be able to adapt easily to the case study in hand, i.e., in the specific application domain. Boilerplates were created to solve problems such as the ambiguity of natural language. The adaptation of the boilerplates (or the creation of new ones, where needed) was implemented in order to respond and fit better in the domain as in EARS for example there are boilerplates (such as trigger) that are specialized for specific fields such as aero engine. Below we present the most popular boilerplates and then we describe our own boilerplates.

2.1.1 Pohl and Rupp

Pohl and Rupp [38] created the following templates by making the following assumptions. Initially, they refer to the obligation of the claims (shall, should, will, may), i.e., to the degree of obligation requirement and identify the following categories: legally obligatory, urgently recommended, future, and desirable. Then they determine the

functionality of each requirement with <process>. For example, the system stores, prints, etc. They also described and distinguished the activity of the system in three categories: a) Autonomous system activity, for example the system performs its functionality autonomously (process verb), b) user interaction, e.g. the system executes its functionality (process) and provides its functionality as a service to the user (provide), and c) the Interface requirement, e.g. the system executes its functionality based on some external event (be able to), i.e. the system must react when it receives data or messages from another system. They also added the object as some verbs have one or more objects and the additional objects that refer to details about the object. Finally, they added the conditions when the system functions are performed under some logical or time constraints. Figure 1 shows the diagram of the boilerplates of Pohl and Rupp [38].

2.1.2 EARS Boilerplates

Next, we describe another known type of boilerplates, namely EARS. In EARS [31], [32], templates are divided into a) ubiquitous requirements: they do not have a condition; they consist of the simplest form, b) event-driven requirements: they start with the word when and are used when a trigger is detected, c) unwanted behavior requirements (if then) are used to declare undesirable situations, d) state-driven requirements (while) are used when the requirements are active for a specified state, or e) optional feature requirements (where) are used when optional attributes are displayed. Figure 2 shows the diagram of boilerplates by [4].

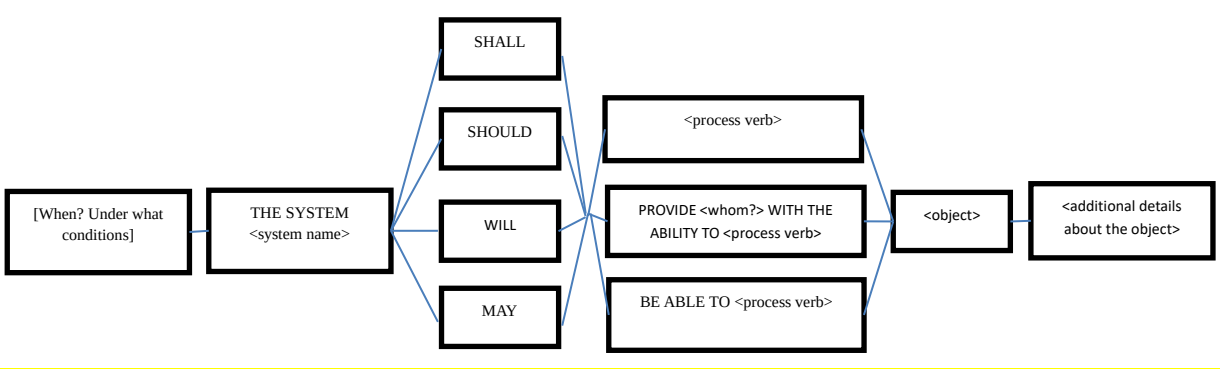

Figure 1. Boilerplates of Pohl and Rupp

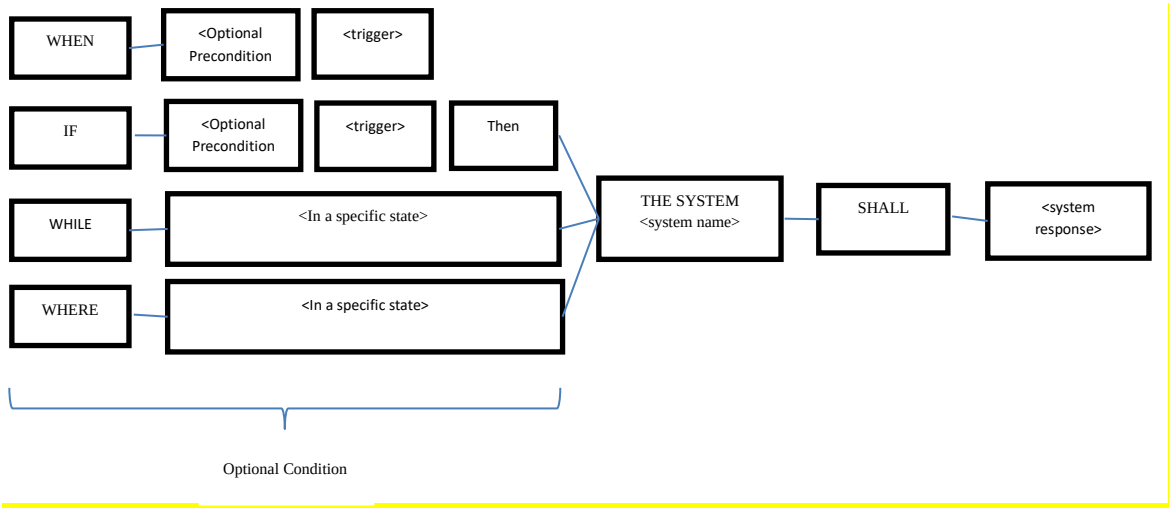

Figure 2. EARS Boilerplates

2.2Boilerplates and NLP

In [3], [4], and [5], authors propose an automated approach that uses NLP, in particular Text Chunking, to check that the templates matched or complied with the requirements correctly. The manual control is time consuming. Text Chunking has proved to be a powerful tool for the process of checking compliance or matching the requirements of natural language with boilerplates.

Zichler & Helke [48] present a new method that converts natural language requirements into boilerplates. They presented the Requirements to Boilerplates Converter (R2BC), which is a flexible and efficient tool to translate the natural language to boilerplates. This transformation constitutes a preparatory stage for machine-readability. For the conversion, NLP is used. Also, this tool, and therefore, the conversion of the requirements to boilerplates, is able to help in subsequent processes, as for example in an automated delta analysis.

Haris & Kurniawan [23] extract requirements from SRS documents using NLP and boilerplates templates. They present an automatic conversion to deal with the disadvantages arising from the manual process, trying to separate requirement sentences and non-requirement sentences. The manual process is tedious and error-prone, it requires much time and involves (usually expensive) experts.

Panthum & Senivongse [37] suggest an automated approach of extracting useful information, from user reviews for mobile, because the manual process is tedious and time consuming. This approach facilitates the development team in the maintenance and evolution process of extracting requirements from user reviews on the App Store and Play Store. Also, machine learning (ML) and NLP are included in this approach. In the first step, text classification algorithms classify user reviews, which hold functional requirements. In the second step, clustering techniques and text similarity identify distinct user reviews because many reviews may relate to the same issue. In the last step,

functional requirements are extracted, making use of patterns and requirement boilerplates.

Usually, Natural Language Processing (NLP) is used in boilerplates (or templates) to convert requirements to boilerplates in order to reduce the conversion process time. The automatic process is faster, while the manual process is time-consuming, tedious and error-prone. Likewise, the transformation of requirements to boilerplates with the use of natural language processing can help in other processes such as in automatic delta analysis.

As far as, our work is concerned, we convert the requirements into boilerplates by making use of the ontology, to avoid the time-consuming process presented by the manual process. The whole boilerplate is stored in the ontology, taking advantage of the natural language syntax similarity between RDF and boilerplates, and the attributes and fixed elements of boilerplates are part of the ontology.

2.3Boilerplates and ontologies

Böschen et al., [7] convert the requirements expressed in natural language into formal specifications. The boilerplates and a knowledge base play important role in the process of conversion. In order to bridge the specific gap between natural language requirements and formal specification they presented a tool, the Requirement Quality Suite, that implements the above idea. These additional structures as boilerplates and the knowledge base are suitable for automatic or semi-automatic analysis of requirements.

Another work by Mahmud et al., [29] incorporated the use of boilerplates in a different way. Multileveled architectural abstractions are used to develop automotive systems in order to manage the great complexity faced by these systems. The authors point out that the disadvantage of boilerplates requirements as well as pattern-based specifications is that they are not capable of supporting and structuring requirements at multiple levels of abstraction, and it is not possible to provide an analysis of these requirements at an early stage. However, they are able to deal with problems arising from the use of natural language (such as ambiguity). For this reason, they proposed a language (Resa) which helps to define boilerplates.

Mokos & Katsaros [33] emphasize the importance of requirements as they determine the operation of a system. The documentation of the requirements in natural language results in ambiguity. The official specification addresses the ambiguities, the underspecified references and the assessment of the requirements, i.e. if they are valid, correct, and feasible. Formalisation and validation of requirements are the basic and initial elements in the development of a system which reduce the verifications and high-cost modifications in later phases in the development of a system. Specifically, they focus on a) the creation of an ontology for a specific domain based on the requirements of the system's specifications, b) the analysis of pattern-based specification languages to address the

ambiguities, and c) the derivation of formal properties from requirements. Finally, they mention challenges of requirements analysis tools.

Stachtiari et al., [42] state that timely verification is needed to avoid high-cost testing and modifications in the final stages of system development. Their work presents a process to address the ambiguity of specifications and guided production of formal properties. Specifically, they presented the formalization and validation of requirements at an early stage using boilerplates and ontology. The process is performed semi-automatically with the help of validation tools. The requirements of their use case come from the field of satellites.

Daramola et al., [11] emphasize the use of ontology combined with the use of boilerplates to strengthen the requirement engineer in documenting high quality requirements (more specifically security requirements) and reduce the workload during the process of requirement specifications. They also state that their approach is a good guide especially for inexperienced requirement analysts.

The above works, in order to specify requirements without ambiguities, combine boilerplates and an ontology to automatically or semi-automatically convert the requirements from natural language into formal specifications. This combination is later used for requirements validation. Furthermore, this combination is a useful tool for non-experienced requirements analysts.

As for our work, we use the combination of boilerplates and ontologies to deal with the ambiguity of natural language and create high-quality, unambiguous requirements. The contribution of our work and the difference with other works is that we took advantage of the natural language syntax of boilerplates and matched it with the syntax of RDF triples. We also present a unique set of boilerplates for our method, that is a unique combination of previous works. Specifically, we created a small set of boilerplates for a specific application domain and a domain-specific ontology. Boilerplates consist of fixed and attributes elements, which the requirements engineer fills in manually or with the help of the ontology after being stored in the ontology. In our work, we saved the whole boilerplate, i.e. both the fixed and attributes elements.

2.4Boilerplates, ontologies and NLP

Farfeleder et al., [15] created a tool that semi-automatically converts natural language requirements into boilerplates. Ontology and techniques used in NLP were applied to this conversion. The tool greatly reduces the effort required manually for conversion. [16] created a semantic guide to help and support requirements engineers in creating semi-formal requirements. The semantic system using the ontology suggests options for the requirements engineer to create the requirements. The system uses the ontology and a set of requirements to provide the options to the user-requirement engineer.

Kravari et al., [26] state the importance of documenting the requirements for developing a system whether it is software, hardware, or embedded systems. Documentation of requirements also greatly affects the success or failure of the system. Despite the efforts

that have been made, the documentation of requirements continues to be an open issue that seeks methods to address. Finally, they propose a new approach based on semantics, ontology, boilerplates, and NLP techniques. Their ultimate goal is to create a framework which incorporate the appropriate boilerplates to identify the requirements and perform the verifications. Moreover, the work of Kravari et al., [27] developed a framework-tool, SENSE, which helps to document the requirements using semantics, ontology, NLP techniques and boilerplates. SENSE was developed to address the difficulties arising from documenting requirements such as ambiguity, speculation, which can also lead to improper operation of the system, which is very difficult to correct later. More specifically, SENSE incorporates a set of boilerplates and proposes the appropriate boilerplate according to the type of requirement and the type of system. Also, SPARQL (SPIN) queries are used to verify the requirements.

In similar related works, ontology, boilerplates and natural language processing are used to convert the requirements into natural language in order to reduce the effort of the manual process. Also, the systems created suggest the appropriate boilerplate to the requirements engineer.

In our work, we make use of boilerplate and ontology to deal with disadvantages arising from natural language, such as ambiguity. The difference with the above works is that we matched the boilerplate with the RDF triples as their syntaxes have a similar form. Also, usually only the attributes are stored in the ontology while we stored the whole boilerplate, namely both fixed and attributes elements.

Our work differs from the previous ones in the following point: we present a combination of an ontology and boilerplates based on the linguistic nature of RDF triples, which is similar to the boilerplate syntax. Boilerplates help reducing the ambiguity of natural language when documenting requirements. They have some fixed words and some attributes. Typically, these attributes are filled by the requirement engineer manually or can be filled with the help of the ontology. Both RDF and boilerplates have a similar syntax, in the form of subject-predicate-object triples. Currently, RDF is the widely accepted W3C standard for the Semantic Web, so there is no reason to consider non-standardized semantic data modeling approaches. Taking advantage of this syntax, we incorporated all of the boilerplates into the ontology and not just the blanks (attributes) to be filled.

In the above works, such as Böschen et al., [7], Mahmud et al., [29], Mokos & Katsaros, [33], Daramola et al., [11], Farfeleder et al., [15], and Kravari et al., [26], the use of boilerplates along with an ontology is observed. These works take advantage of the combination of ontology and boilerplates in order to define the requirements. But only the completion of the attributes of boilerplates is carried out or completed by the ontology. The contribution of our work lies in the fact that the whole boilerplate is stored in the ontology and not just the attributes (which usually have to be filled in to use the boilerplates). Specifically, both fixed elements and attributes are part of the ontology. While in the above related works only attributes are part of the ontology. Usually, these attributes are filled by the requirement engineer manually or can be filled with the help of

the ontology. This method results in semantically correct requirements being created. The above aspect is the novel contribution of our work, compared to the rest of the relevant literature.

In more detail, in this work we have created our own boilerplates based on Pohl and Rupp [38] and EARS Boilerplates [31], [32], as well as our own ontology for the ATM field. We exploit the linguistic nature of Resource Description Framework (RDF) triples and create our boilerplate syntax. RDF describes resources and the relationships between them in triplet form. It is a data model. RDF triplet contains statements such as subject - predicate (verb) – object. This statement is very similar to the boilerplate language. Our boilerplates fall into two categories: a) basic boilerplate template and extended form of basic boilerplate template and b) and boilerplate template with temporal or logical conditions.

As far as it concerns the industrial application areas, we have observed that boilerplates were applied to aero engine control system requirements [4], [7], [15], [16], [31], [32], [43], [44], [45] or consequently in satellite component [5], security requirements [9], [10], [11] on requirements for automotive systems [29], [30]. In this article the requirements we use come from the ATM domain. In order to express all the requirements of this domain, we created new boilerplates based on adapting pre-existing boilerplates from other domains.

3. Ontology and requirements of the ATM use case

Gruber [21] and [22] refer that an ontology is "an explicit specification of a conceptualization". Antoniou et al., [1] mention that the ontology is "a model of a particular domain built for a particular purpose". Ontologies represent the knowledge of a domain, defining domain entities and relations among them. Below, this paper presents the ontology we created for the ATM domain such as classes, hierarchies, object properties, etc. using the Protégé ontology editor [35]. Also, we created the images from Protégé. A study of the field (ATM) was carried out for the representation of knowledge, and we used the general Ontology Development 101 methodology [36]. The domain we have used is an ATM, with a well-known set of requirements. Regarding the ATM description, it is worth noting that there the most important concepts in the domain are Account, ATM, Bank, Bank Computer, Cash Card, Customer, and Transaction. The Account is defined in the ontology as a subclass of the Product class, which products are offered by the bank. The account is used for transactions and comes in several varieties, including Current Account and Saving Account (Figure 3). A customer may have multiple accounts. Figure 4 depicts the relationship between Account and Customer.

The ATM allows the customers through a CashCard to make transactions. Essentially the ATM is the inter-mediary or the link between the customer and the Bank-Computer. More specifically, it accepts the customer's selections and transfers them to the bank computer to process and approve them, and then to take the appropriate action, such as cash dispensing. There is an interaction between the ATM and the customer and between ATM

and bank computer. In the ontology we set the ATM entity as a subclass of the Service class offered by Bank (Figure 3). Figure 4 shows some relations between ATM and other entities, such as ATM accepts CashCard, ATM waits ResponseFromBankComputer, ATM returns CashCard, ATM dispenses Money. The Bank is a financial institution that includes customers, accounts and ATMs through which customers can make transactions as shown in Figure 5.

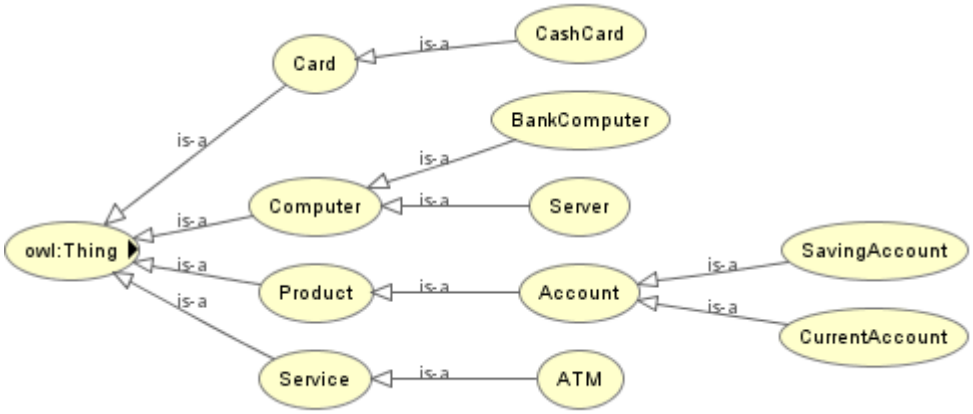

Figure 3. Hierarchy of the main classes of the ontology.

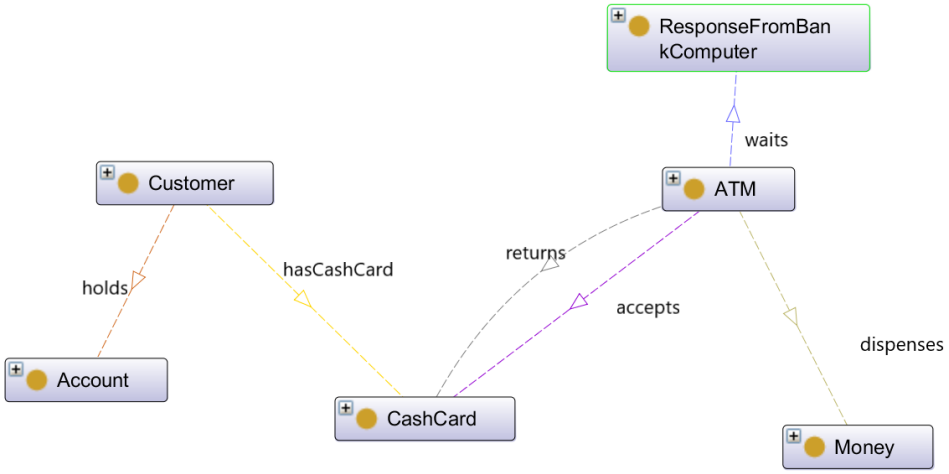

Figure 4. Relationships among the main classes of the ontology

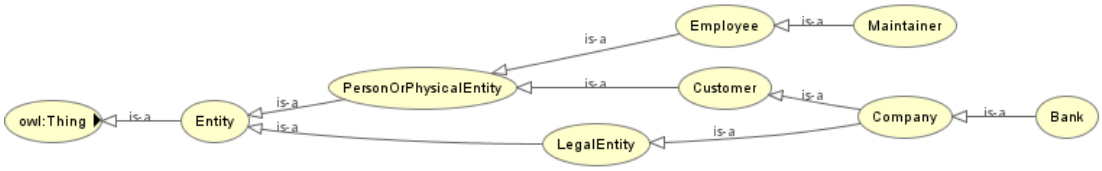

Figure 5. Subclasses of the class Entity

Bank computer (class BankComputer) is the computer that interacts with the ATM and the bank cashiers. A bank may have a complete computer network; however, here we focus on the one that interacts with the ATM and the bank cashiers. In the ontology of this paper, we set the BankComputer entity as a subclass of the Computer class used by Bank (Figure 3).

A cash card (class CashCard) belongs exclusively to a bank customer and allows authorized access to accounts via the card. Each cash card has a bank code (bankcode) and cash card number (cardSerialNumber) that identifies the bank code and the acccounts that the card can access, respectively. A cash card may not have access to all the customer accounts. Also, multiple copies of the card may exist; for this reason, the simultaneous use of the card by different machines must be considered. In the ontology, we set the CashCard entity as a subclass of the Card class because it is possible to have many types of cards (Figure 3); in these specific use case this paper focus on the cash card for dispensing money. Also, bankcode and cardSerialNumber are data properties. Figure 4 shows some relations between CashCard and other entities such as ATM accepts CashCard, Customer hasCashCard CashCard.

The customer can have more than one accounts. Also, a customer may consist of one or more persons or organizations. In the ontology, we set the Customer entity as a subclass of the PersonOrPhycicalEntity as shown in Figure 5.

A transaction is a request to perform certain functions on a customer's accounts. This use case requirements focus on cash dispensing. In the ontology the Transaction entity has the subclasses shown in Figure 6.

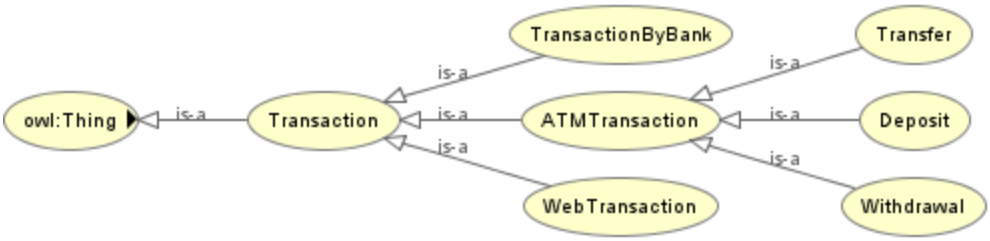

Figure 6. Subclasses of the class Transaction.

The ATM does not operate individually and depends on and cooperates with the bank's computers. The operation of the ATM is as follows: a) it accepts the cash card of customer, b) a dialogue between the customer and the ATM takes places, c) the ATM communicates with the bank computer to carry out the transactions, and d) it dispenses cash and receipts.

According to the requirements specifications, the requirements are divided into functional requirements, interface requirements, performance requirements and other requirements. The largest percentage of requirements is occupied by the first category, the functionals requirements which are divided into ATM requirements and bank computer requirements.

The requirements, such as interface and other requirements, are formulated in a vague way; therefore, they could not be used accurately in conjunction with some external entities to which they refer. An example of such requirement is*: "The ATM network has to provide hardware interfaces to various printers, various ATM machines (There are several companies producing the ATM machines), several types of networks (The exact specification of the hardware interfaces is not part of this document)".*

Some examples of functional requirements of ATM are:

If no cash card is in the ATM the system should display initial display.

If the ATM is running out of money no card should be accepted. An error message is displayed.

Some examples of functional requirements of bank computer are:

The bank computer checks if the password is valid for a valid cash card.

If it is a valid cash card and a valid password but there are problems with the account, the bank will send a message to the ATM that there are problems.

In this paper, the above examples and their corresponding conversions to our boilerplate scheme are found in Table 3.

Some examples of performance requirements: "*Error message should be displayed at least 30 sec*", "*if there is no response from the bank computer after a request within 2 minutes the card is rejected with an error message*". The above examples and their corresponding conversions are found in Table 2 and Table 3.

4. The Boilerplates of our method

In this section, we describe the conversion of requirements into boilerplates. The domain we have used is an ATM, with a well-known set of requirements. The presentation of the following subsections is based on the different types of boilerplates, accompanied by one or more examples of requirements represented using each boilerplate type. Notice that a requirement may consist of several sentences and each sentence may correspond to a different template category. Below, we give examples of requirements and the corresponding boilerplates using the ontology. Our methodology is applied when the requirements engineer is not allowed to choose other values for the entities except for those related to the domain/range of the properties, i.e. the verbs that connect the requirements. Also, we mention that our method restricts importing entities only to those that can be semantically related through the selected verbs, preventing semantic errors by design. Our goal was to represent all the boilerplates into the ontology as sets of RDF triples (semantic graphs), starting with the basic boilerplate as a single triple and gradually extending it with more complex boilerplates. Some pre-existing boilerplates may match the syntax /

semantics of the ATM requirements, but some would need to be adapted in order to able to capture more complex relationships among ontology entities which are important to be represented independently in the domain of the ATM. In several cases it is necessary to remodel the existing boilerplates to suit the new field or requirements. In all pre-existing boilerplates there is the following assumption "subject verb object"; we modelled this basic boilerplate syntax as an RDF triple, exploiting the fact that is identical to the RDF model. We also extended this basic structure with logical and temporal conditions, as well as secondary sentences and attributes that characterize the subject or the object of the sentence. These more complex boilerplates are modelled as sets of connected RDF triples, a.k.a. semantic graphs (nets).

Thus, there was a need to generate multiple boilerplates, more complicated than the basic one, in order to be able to represent more complex relationships among the model entities that need to be represented in the ontology and later to be used for semantically analysing the requirements for completeness, correctness, etc. Just the basic boilerplate cannot deal properly with all these complicated relationships. Table 1 shows the differences and similarities between the EARS boilerplates, boilerplates of Pohl and Rupp and our boilerplates.

4.1 Basic Boilerplates

The core of the above boilerplates is a subject-verb-object syntactic template. With respect to the boilerplates of Pohl and Rupp, we observe that the subject is the system (<*system name*>) and the basic core is functionality, which is referred to as a process (<*process verb*>) and is always determined using verbs. Simultaneously with the determination of the process, the activity of the system is defined. For example, if the system performs its functionality (process) autonomously, then the verb is not accompanied by something else. If the system interacts with the user, then the following provide <*whom?*> with the ability to accompanies <*process verb*>. If the system interacts with other systems, then the following be able to accompanies <*process verb*>. Because some verbs require one or more objects to have a complete meaning, the <*process verb*> is accompanied by <*object*>. Also, depending on the priority (obligation) of a requirement, such as obligatory requirements, urgently recommended requirements, future requirements, and desirable requirements, the <process verb> can be characterized correspondingly with modal verbs shall, should, will, and may. A complete requirement template without conditions looks like the one in Figure 5, without the conditions in the left. Similarly, this boilerplate is equivalent to the general form of the EARS boilerplates without preconditions, i.e., <*system name*> *shall* <*system response*>. A complete requirement template without conditions looks like the one in Figure 6 without the optional conditions in the left. For example, in the requirement "The control system shall prevent engine overspeed", we observe that the verb and the object correspond to <*system response*> (or the object takes the place of <*system response*> and the verb accompanies shall), while the boilerplates of Pohl and Rupp had them separately. Another difference is that in EARS boilerplates there is no priority (obligation) of the verbs.

Concerning our general form without preconditions and conditions, it follows the linguistic nature of RDF triples that is similar to the syntax of the boilerplates. The structure of RDF (Resource Description Framework) statements is independent of the domain data and is based on the triple: entity-attribute-value. In this paper we use the combination of ontologies and boilerplates, and the creation of the boilerplates was based on the triplets of RDF *<subject> <relation> <object> or <subject> <verb> <object>*. The *<subject>* corresponds to <system name> of the Pohl and Rupp and EARS boilerplates. The *<relation>* will be referred as <verb> in our general form and corresponds to the <process verb> of the Pohl and Rupp boilerplates and the word shall of the EARS boilerplates. The *<object>* is the object of the *<verb>*.

In this paper, we call basic boilerplate our general template form *<subject> <verb> <object>*. The requirements that adhere to the basic boilerplate complete the various parts of the template by selecting instances from the ontology. An example of natural language ambiguity resolved by the boilerplate method is given below. The object property returns according to its domain and range restrictions should be filled in as follows: *<ATM> <returns> <CashCard>*. While without the ontology it could be completed as follows: *<Customer> <returns> <CashCard>*, which is semantically wrong. Furthermore, the requirements themselves are stored in the ontology as instances of the corresponding boilerplate class. Also, in this paper we have not added to the general form priorities for verbs such as boilerplates of Pohl and Rupp but if the engineer wishes she can add them in the form of comments [] such as *<subject> [shall] <verb> <object>*. The requirement engineer can add comments as [details] but these are not stored in the ontology. The details can be added in any position. The basic syntax of boilerplate is "subject verb object". If the requirement engineer wants for his own convenience to write details before or after or even between the components of a boilerplate, then the engineer can do so in the form of comments. In the current work we did not give the verbs priorities- (shall, should, will, and may such as Pohl and Rupp) but it could be done. So, if we have the following boilerplate <ATM> <displays> <has_no_money> someone write can it as follows: <ATM> [should] <returns> <CashCard >. In the comments, the engineer can indicate priorities, but everything that is also needed later. Comments, of course, are not taken into account for the semantic analysis. The position of details can be embedded anywhere in a boilerplate. The details can also be integrated into all categories of boilerplates. Examples of positions for details for basic category (basic boilerplate) are as follows: Some examples are: *[details] <subject> <verb> <object>, <subject> [details] <verb> <object>, <subject> <verb> [details] <object>, <subject> <verb> <object> [details], [details] <subject> [details] <verb> [details] <object> [details]*. Also, there are examples in Table 6.

The basic template, *<subject> <verb> <object>*, is used in sentences that are usually in active voice. However, in most cases a requirement in passive voice can be transformed into an active voice, so that it can be mapped into the basic boilerplate, as well. Boilerplates are in the active voice in order to force engineers to use it instead of the passive which introduces ambiguities. In this paper, we followed this approach in order to minimize the number of boilerplate types, which is less confusing for the final user. Below (Table 2), we present some example requirements and their corresponding conversion to

the basic boilerplate. In case a verb has more than one objects, then the general form becomes as follows *<subject> <verb> <object>+* and the engineer uses the specific boilerplate, which is an extension of the general form. The example 8 in Table 2 uses a verb that accepts many objects.

In terms of interaction between systems, or interface requirement as mentioned in boilerplates of Pohl and Rupp, requirements define how a system performs an activity that is dependent on other systems. For example, when the system receives a message and has to perform a specific function or behavior. An appropriate template for interface requirements is:

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

In this paper, we noticed that the interaction between the systems-computers, at least in the ATM case, occupies a large percentage and we thought that we should create the following "interaction" boilerplates. It is essentially an extension of the basic template *<subject> <verb> <object>* that we have and is based on information retrieved from the ontology. Examples of such interaction requirements are given above in Table 2 such as example 8. Also, comments are added by the user with the form of [details]. The details can be added in any position. The syntax of the interaction requirement boilerplates is:

<subject> <sends> <object> To <entity>

<subject> <receives> <object> From <entity>

Also, we have observed that many verbs have as an object, a property of a specific entity instead of an entity. This sometimes happens for the verb subject, as well. For this reason, we extended the basic template *<subject> <verb> <object>*. Examples of such boilerplates are given above in Table 2, such as example 5, 6, 7, 8, 9 and 11. Also, [details] which are formed by the user could be added, which are explanatory comments regarding the specifications. The details can be added in any position. The syntax of these boilerplates is:

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

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

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

Table 1. Comparison of EARS, Pohl and Rupp and our Boilerplates

Categories	**Boilerplates of Pohl and Rupp.**	**Categories**	**EARS Boilerplates.**	**Categories**	**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>*

Categories	**Boilerplates of Pohl and Rupp.**	**Categories**	**EARS Boilerplates.**	**Categories**	**Our Boilerplates**
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>* *<subject> of <entity> <verb> <object>* *<subject> <verb> <object> of <entity>* *<subject> of <entity> <verb> <object> of <entity>* *<subject> <verb> <object> + From <entity> <subject> <verb> <object> + To <entity* *<subject> <verb> <object> for <numerical-comparison-operator> <number> <TimeUnit>*
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 + Extended logical condition boilerplate*

Categories	Boilerplates of Pohl and Rupp.	Categories	EARS Boilerplates.	Categories	Our Boilerplates
					Nested if: *if basic boilerplate + then* *if basic boilerplate + then* *basic boilerplate+* *Temporal Condition boilerplate(After or When)* *After basic boilerplate, basic boilerplate* *When basic boilerplate, basic boilerplate*
		Unwanted behaviours	*IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>*		

Table 2. Examples of Basic Boilerplates

No.	Requirements expressed in natural language	Requirements expressed using the Basic Boilerplate	Category Boilerplate
1.	Return cash card.	*<ATM> <returns> <CashCard >*	*<subject> <verb> <object>*
2.	Display an error message (ATM has no money).	*<ATM> <displays> <has_no_money>*	*<subject> <verb> <object>*
3.	The ATM has to check if the entered card is a valid cashcard.	*<ATM> <checks> <CashCard >*	*<subject> <verb> <object>*
4.	A card is entered.	*<Customer><enters> <CashCard >*	*<subject> <verb> <object>*
5.	The serial number should be logged.	*<BankComputer><records> <cardSerialnumber> of <CashCard>*	*<subject> <verb> <object> of <entity>*
6.	Initialize parameter k. The k is the maximum withdrawal per day and account	*<Maintainer> <initializes> <accountMaxWithdrawalPerDayAndAccount > of <Account>*	*<subject> <verb> <object> of <entity>*
7.	The amount of cash is less than t.	*<transactionAmount> of <Transaction> <is_less_than_or_equals_to> <accountMaxWithdrawalPerDayAndAccount > of <Account>*	*<subject> of <entity> <verb> <object> of <entity>*
8.	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>*
9.	Read the serial number.	*<ATM> <reads> < cardSerialnumber > of <CashCard>*	*<subject> <verb> <object> of <entity>*
10.	Receive response from bank (about	*<ATM> receives <rejectionAutorization> From <BankComputer>*	*<subject> <verb> <object> From <entity>*

No.	Requirements expressed in natural language	Requirements expressed using the Basic Boilerplate	Category Boilerplate
	authorization).		
11.	The user is requested to enter his password.	*<Customer> <types> <password> of <Customer>*	*<subject> <verb> <object> of <entity>*
12.	Error message should be displayed at least 30 sec	*<ATM> <displays> <Display> for <atLeast> <30> <Second>*.	*<subject> <verb> <object> for <numerical-comparison-operator> <number> <TimeUnit>*

4.2 Conditional and temporal boilerplates

Pohl and Rupp [39] refer that the functionalities are usually not continuous but are applied under certain conditions, which are distinguished into logical and temporal. The conjunctions as soon as and if are used for time and logic conditions, respectively. When the condition is not clear whether it is a temporal or logical, then the conjunction when is used. The conditions are used in the beginning of a requirement as shown in Figure 1. For the EARS boilerplates, the conditions are divided as follows: event-driven requirement, unwanted behaviors, state-driven requirement, and optional feature requirement. An event-driven requirement is needed when a triggering event occurs, and the corresponding template is: *WHEN <optional preconditions> <trigger> the <system name> shall <system response>*. An example of this is *"When continuous ignition is commanded by the aircraft, the control system shall switch on continuous ignition"*.

An unwanted behavior requirement is similar to the previous one, the difference being that it is used only for unwanted situations; the conjunctions *if* and *then* are used. The corresponding template of this type is: *IF <optional preconditions> <trigger>, THEN the <system name> shall <system response>*.

A state-driven requirement is used while the system is in a defined state. The corresponding template of state-driven requirement is: *WHILE <in a specific state> the <system name> shall <system response>*.

Instead of the word *while* the word *during* can be used for easier reading of the requirement. The last condition, the optional feature requirement, is used when the system includes a feature. The corresponding template is: *WHERE <feature is included> the <system name> shall <system response>*. In terms of our own conditions' boilerplates, we distinguish two categories: conditional boilerplate and temporal boilerplate. So, in this paper we have introduced the conditional boilerplate. We use the keyword *if* and *then*. Also, in this template we use the binary operators *and*, *or*. Such a template consists of the *condition part* and the *then part*. Essentially, the condition is a logical expression. A condition can have one or more logical expressions connected through binary operators. The *then part* can consist of one or more basic boilerplates. Also, the *condition part* can consist of one or more basic boilerplates. In case of more than one basic boilerplates in the *condition part*, they are connected with binary operators. The syntax of conditional boilerplate is *if basic boilerplate*$^{+}$ *then basic boilerplate*$^{+}$.

In this paper, we also extended the basic template of the conditional boilerplate (*if basic boilerplate then basic boilerplate*) to consider cases in which if the condition does not apply to do something *else*. For this reason, we added the keyword *else*. The syntax of extended conditional boilerplate is *if basic boilerplate* $^{+}$ *then basic boilerplate* $^{+}$ *else basic*

boilerplate $^{+}$. It is also possible to have a condition within the condition (nested if). In Table 3 there is such an example. After the *else* keyword more than one basic boilerplate can exist. Comments in the form of *[details]* could be added by the requirement engineer. The details can be applied to any position. In summary, for conditional statements we can have one template for the basic conditional boilerplate and one template for the extended conditional boilerplate. Below (Table 2), we present some example requirements and their corresponding conversion to the conditional boilerplates. Templates / boilerplates reduce the ambiguity which is contained in the natural language. However, nested conditions or loops are used to formulate specific requirements within the template-boilerplate. Since a concrete syntax is enforced, ambiguity is resolved via nesting and scoping of conditions within one level. The corresponding syntax is as follows:

Basic conditional boilerplate:
if basic boilerplate $^{+}$ *then basic boilerplate* $^{+}$
Extended conditional boilerplate:
if basic boilerplate^{+} *then basic boilerplate* $^{+}$
else
basic boilerplate $^{+}$
Nested if:
if basic boilerplate $^{+}$ *then*
  if basic boilerplate $^{+}$ *then*
    basic boilerplate^{+}

For the requirements that indicate time-temporal relations, we have chosen the conjunctions *when* and *after* in order to be easily differentiated from the logical conditions. These boilerplates are called *temporal*. The engineer selects the appropriate conjunction according to the meaning of the requirement. Such a temporal template consists of the *temporal sentence* and the *main sentence*. A *temporal sentence* can consist of one basic boilerplate. The *main sentence* can also consist of one basic boilerplate. Also, *[details]* could be added, which are explanatory comments regarding the specifications. The details can be applied to any position. Below (Table 4), we present some example requirements and their corresponding conversion to the temporal boilerplates. The syntax of the temporal boilerplates is as follows:
After basic boilerplate, basic boilerplate
When basic boilerplate, basic boilerplate

Table 3. Examples of Conditional Boilerplates

No.	Requirements expressed in natural language	Requirements expressed using Conditional Boilerplates	Category Boilerplate
1.	If no cash card is in the ATM, the system should display the initial display.	*if* <*CashCard*> <is_in_not> <*ATM*> *then* <*ATM*> <displays> <*Initial Display*>	*if* <*subject*> <*verb*> <*object*> *then* <*subject*> <*verb*> <*object*>
2.	If the ATM is running out of money, no card should be accepted. An error message is displayed.	*if* <*Money*> <is_in_not> <*ATM*> *then* <*ATM*> <not_accept> <*CashCard*> <*ATM*> <displays> <*has_no_money*> This could be done as follows, as well: *if* <*ATM*> <*is_in_state*> <*No_Money*> *then* <*ATM*> *[should]* <*not accept*> <*CashCard*> <*ATM*> <*displays*> <*has_no_money*>	*if* <*subject*> <*verb*> <*object*> *then* <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*>
3.	Check if it is a valid cash card. It will be valid if the	if <*ATM*> <reads> <*cardSerialnumbe*r> of <CashCard> then	*if* <*subject*> <*verb*> <*object*> *of* <*entity*> *then*

No.	Requirements expressed in natural language	Requirements expressed using Conditional Boilerplates	Category Boilerplate
	information on the card can be read, and it is not expired.	if <CashCard> <is_in_not_state> <expireState> then <CashCard><is_in_state> <StatusValidCashCard>	*if <subject> <verb> <object> of <entity> then* *<subject> <verb> <object>*
4.	The bank computer checks if the password is valid for a valid cash card. Input: Request from the ATM to verify password. Processing: Check password of the customer. Output: Valid or invalid password	*<ATM> <sends> <password> of <Customer> To <BankComputer>* *if <typedpassword> of <Customer> not_equals <password> of <Customer> then* *<CashCard><is_in_state> <StatusBadPassword>* *<CashCard> <is_in_state> <StatusInValidCashCard>* *<BankComputer> <sends> <rejectionAuthorization> <StatusBadPassword > To <ATM>* *else: <CashCard> <is_in_state> < StatusValidPassword><CashCard> <is_in_state> < StatusValidCashCard> <BankComputer> <sends> <acceptAuthorization> <StatusValidPassword> to <ATM>*	*<subject> <verb> <object> of <entity> + To <entity>* *if <subject> of <entity> <verb> <object> of <entity> then* *<subject> <verb> <object>* *<subject> <verb> <object>* *<subject> <verb> <object>+ To <entity>* *else:* *<subject> <verb> <object>* *<subject> <verb> <object>* *<subject> <verb> <object>+ To <entity>*
5.	If password and serial number are ok_ the authorization process is finished.	*if <CashCard> <is_in_state> <StatusValidPassword> <CashCard> <is_in_state> <StatusValidSerialNumber> then* *<Authorization> <is_in_state> <FinishedAuthorizationState>*	*if <subject> <verb> <object>* *<subject> <verb> <object>then* *<subject> <verb> <object>*
6.	If it is a valid cash card and a valid password but there are problems with the account the bank will send a message to the ATM that there are problems	*if <CashCard> <is_in_state> < StatusValidPassword> <CashCard> <is_in_state> < StatusValidBankCode> <CashCard> <is_in_state> < StatusBadAccount> then* *<BankComputer> sends <rejectAuthorization> <StatusbadAccount> To <ATM>*	*if <subject> <verb> <object>* *<subject> <verb> <object>* *<subject> <verb> <object>* *then* *<subject> <verb> <object>+ To <entity>*

4.3 Complex boilerplates

This kind of boilerplates incorporates complex sentences. In this paper, a complex sentence consists of a main sentence (in our case a basic boilerplate) and a secondary sentence (in our case also a basic boilerplate). The subordinate clause is the dependent clause and cannot stand alone. Complex boilerplates describe requirements that cannot be described by just one basic boilerplate, but they are more complex requiring the combination of many basic boilerplates as in the case with the temporal and logical conditions, which we characterize as complex boilerplates, too. Table 5 reports some examples of complex boilerplates. The entities are enclosed in <> while keywords are not. Keywords are the following: if, then, To, From, else, After, While, of, and. After, While, of, and. The system has ready templates/boilerplates and the user chooses one of them to fill it with entities from the ontology. Of course, templates correspond to a subset of natural language.

Table 4. Examples of Temporal Boilerplates

No.	Description of requirements	Boilerplates	Category Boilerplate
1.	The authorization starts	After <*Customer*> <enters> <*CashCard*>,	After <*subject*> <*verb*> <*object*>,

No.	Description of requirements	Boilerplates	Category Boilerplate
	after a customer has entered his card in the ATM.	<*Authorization*> <is_in_state> <startState>	<*subject*> <*verb*> <*object*>
2.	If the cash card is valid the ATM should read the serial number and bank code.	After <*CashCard*> <is_in_state> <*ValidCashCard*>, <*ATM*> [should] <read> <*cardSerialnumber*> of <CashCard> <*ATM*> [should] <read> <*bankCode*> of <Bank>	After <*subject*> <*verb*> <*object*>, <*subject*>*[details]* <*verb*> <*object*> *of* <*entity*> + *To* <*entity*> <*subject*> *[details]* <*verb*> <*object*> *of* <*entity*> + *To* <*entity*>
3.	The kind of transactions the ATM offers is Withdrawal Input: Authorization successfully completed. Enter the amount to withdraw Processing: Amount entered is compared with m. Output: Amount of money to be dispensed is displayed. Begin initial withdrawal sequence	*After* <*Authorization*> <is_in_state> <*FinishedState*>, <*ATM*> <displays> <*ChoiseOfTranscaction*> <*Customer*> <types> <*transactionAmount*> of <Transaction>. if <*Money*> <*is_less_than_or_equals_to*> <*bankMaxWithdrawalPerTransaction*> *of* <*Bank*> *then* <*ATM* > <returns> <*CashCard*> <*ATM* > <dispenses> <*Money*>	After <*subject*> <*verb*> <*object*>, <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*> *of* <*entity*> if <*subject*> <*verb*> <*object*> *of* <*entity*> *then* <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*>

Table 5. Examples of Complex Boilerplates

No.	Description of requirements	Boilerplates	Category Boilerplate
1.	Authorization successfully completed. Enter the amount to withdraw	After <*Authorization*> <is_in_state> <*FinishedState*>, <*ATM*> <displays> <*ChoiseOfTranscaction*> <*Customer*> <enters> <*transactionAmount*> of <Transaction>	After <*subject*> <*verb*> <*object*>, <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*> *of* <*entity*>
2.	If the transaction is not successful, an error message should be displayed. The card should be ejected.	If <*ATM*> <receives> <*TransactionNotSucceeded*> From <*BankComputer*> then <*ATM* > <displays> <*has_no_money*> <*ATM* > <returns> <*CashCard*>	if <*subject*> <*verb*> <*object*> *From* <*entity*> *then* <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*>
3.	If it is not a valid bank code the bank computer will send a message to the ATM Input Invalid bank code Processing Process message Output The bank computer sends the message bad bank code to the ATM.	If <*CashCard*> <is_in_state> <StatusBadBankCode> then <*BankComputer*> <*sends*> <*rejectionAuthorization*> <*StatusBadBankCode* > *To* <*ATM*> <*ATM*> *receives* <*rejectionAutorization*> <*StatusBadBankCode*> *[for authorization dialog] From* <*BankComputer*> <*ATM*> <*returns*> <*CashCard* > <*ATM*> <*displays*> <*WrongBankCode*>	if <*subject*> <*verb*> <*object*>*then* <*subject*> <*verb*> <*object*>+*To* <*entity*> <*subject*> <*verb*> <*object*> *[details]*+ *From* <*entity*> <*subject*> <*verb*> <*object*> <*subject*> <*verb*> <*object*>

In order for a requirements' engineer to create a requirement, they must select the appropriate boilerplate from the categories of boilerplates that we created, as shown in Table 1 or Table 10. A boilerplate consists of fixed and attributes elements. The attributes elements are in quotation marks (such as *<subject>, <verb>, <object>*) and fixed elements are not in quotation marks (such as *of, To, From, if, then*). The attributes elements are filled in by the engineer either manually or automatically or semi-automatically. In our methodology, after choosing the appropriate boilerplate, the latter must be completed with attributes elements from the ontology. Table 6 presents examples of categories, the corresponding fixed and attributes elements, as well as completed boilerplates from the ontology. Suppose the engineer wants to generate a requirement for the following requirement in natural language *Return cash card.* The requirements engineer choose the basic category *<subject> <verb> <object>*. Then, the engineer completes the attributes from the ontology and it is done as follows: *<ATM> <returns> <CashCard>*. The RDF triple in the ontology is *ATM returns CashCard*. We exploit the natural language syntax of boilerplates (such as *<subject> <verb> <object>*) mapping them to RDF triples which have also a linguistic nature (such as the RDF triple: *ATM returns CashCard*). First, the search, in ontology, is made for the verb of the natural language requirement statement, specifically in the object properties of the ontology. And by extension, after finding the appropriate object property (for example *returns*), the engineer completes the subject from the domain of the object property (*ATM*) and the object from the range (*CashCard*) of the object property. Table 8 shows a similar example of a basic boilerplate with elements from the ontology.

Table 6 Examples of fixed and attributes elements of boilerplate

Boilerplate	**Fixed elements**	**Attributes elements**	**Example completed Boilerplate with ontology**
The <system> shall be able to <function> not less than <quantity> <object> Hull et al., (2010)	*The, shall be able to, not less than, within*	*<system>, <function>, <object>, <quantity>*	*The <communications system> shall <sustain> <telephone contact> not less than <10> <callers>*. Hull et al., (2010)
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> <displays> <has_no_money>*
<subject><verb><object>of <entity> *Our boilerplate*	*of*	*<subject><sends><object> of <entity>*	*<Customer> <types> <password> of <Customer>*
<subject><verb><object>From <entity> *Our boilerplate*	*From*	*<subject><sends><object> <entity>*	*<ATM> <receives> <rejectionAutorization> From <BankComputer>*
<subject> <verb> <object> of <entity> + To <entity>: *<subject><verb><object>of <entity><object>of <entity>To <entity>* *Our boilerplate*	*of, of, To*	*<subject><verb><object> <entity><object><entity> <entity>*	*<ATM> <sends> <typedpassword> of <Customer> <cardSerialNumber> of <CashCard> To <BankComputer>*
if basic boilerplate + then basic boilerplate + *Specifically:* *if <subject> <verb> <object> then <subject> <verb> <object>* *Our boilerplate*	*if, then*		*if <CashCard>* <is_in_not> *<ATM> then* <*ATM*> <displays> *<Initial Display>*

As we mentioned above, a boilerplate consists of fixed and attributes elements. The fixed elements that we created are as follows: *of, To, From, After, When ,if, then, else*. The following: *<subject>, <verb>, <object>, <entity>* are the attributes elements of our boilerplates. The attributes elements are completed by the requirements engineer in the specific work with the help of the ontology. Usually, the values of the ontology for the attribute *<subject>* can be a class, for attribute *<verb>* can be an object property, and for attribute *<object>* can be either an instance or a class. In the case of *<object> of <entity> or <subject> of <entity>*, the attribute object and subject can be a datatype property. The attribute *<entity>* can be a class. Table 7 shows examples for attributes with values from the ontology and their corresponding description in the ontology.

Table 7 Boilerplates Attributes with Values

Examples completed Boilerplate	**Category Boilerplate**	**Boilerplate Attribute**	**Description**	**Example of Values from Ontology**
<ATM> <displays> <has_no_money>	<subject> <verb> <object>	*<subject> <verb> <object>*	class, predicate (object property), instances	*ATM, displays, has_no_money*
<Customer> <types> <password> of <Customer>	*<subject><verb> <object> of <entity>*	*<subject><verb> <object> <entity>*	class, predicate (object property), data property, class	*Customer, types, password, Customer*
<ATM> <receives> <rejectionAutorization> From <BankComputer>	*<subject><verb> <object> From <entity>*	*<subject><verb> <object> <entity>*	class, predicate (object property), instances, class	*ATM, receives, rejectionAutorization, BankComputer*
<Customer><enters> <CashCard >	*<subject><verb> <object>*	*<subject><verb> <object>*	class, predicate (object property), class	*Customer, enters, CashCard*
<transactionAmount> of <Transaction> <is_less_than_or_equals_to> <accountMaxWithdrawalPerDayAnd Account > of <Account>	*<subject>of <entity> <verb> <object>of <entity>*	*<subject> <entity> <verb> <object> <entity>*	dataproperty, class, predicate (object property), dataproperty, class	*transactionAmount, Transaction, is_less_than_or_equals_to accountMaxWithdrawalPerDayAndAccount, Account*

In Table 7, in the following examples *<Customer> <types> <password> of <Customer> or <transactionAmount>of<Transaction><is_less_than_or_equals_to><accountMaxWithdrawalPerDayAndAccount > of <Account>*, we notice that the object of the verb or the subject of the verb cannot be a class. For example, the following requirement: *The user is requested to enter his password.* In the ontology, the *password* is a datatype property with domain

Customer and cannot be defined as a class, because it is a datatype property. For this purpose, it is useful to define a class *PasswordOfCustomer*. Also, we need to create a new object property, naming it *ofCustomer.* The domain of object property or predicate, *ofCustomer,* is the class *PasswordOfCustomer* and the range is the class *Customer.* So, the data property *password* has domain the class *PasswordOfCustomer*. This class that we created is called metaclass. The category of boilerplate for this requirement (*The user is requested to enter his password*) is the *<subject> <verb> <object> of <entity>*. In the ontology, we store this requirement as an RDF triple, namely as *subject-predicate-object*, such as *Customer types PasswordOfCustomer*. We create the metaclass because the above triple cannot have as object the data property *password*. Table 8 shows an example of a basic boilerplate and Table 9 an extended boilerplate with elements (values) from the ontology. Regarding the *<subject>of <entity>*we followed the same rationale.

Table 8 Example of a basic boilerplate with elements from the ontology

Requirement	A card is entered.
Category	*Basic boilerplate*
Boilerplate	*<subject> <verb> <object>*
Completed Boilerplate	*<Customer><enters> <CashCard >*
RDF triplet in ontology	*Customer enters CashCard*
Object property	*enters*
Domain of object property	*Customer*
Range of object property	*CashCard*

Table 9 Example of an extended boilerplate with elements from the ontology

Requirement	*The user is requested to enter his password.*
Category	*Extended basic boilerplate*
Boilerplate	*<subject> <verb> <object> of <entity>*
Completed Boilerplate	*<Customer> <types> <password> of <Customer>*
RDF triplet in ontology	*Customer types PasswordOfCustomer*
objectOfEntity (class) or range of object property (types)	*PasswordOfCustomer*
Domain of object property (types)	*Customer*
Object property	*types*
Object property	*OfCustomer*
Domain of object property (*OfCustomer*)	*PasswordOfCustomer*
Range of object property (*OfCustomer*)	*Customer*
Domain of data property (cardSerialNumber)	*PasswordOfCustomer*

4.4 Semantic verification of requirements

SPARQL queries are used to ensure the semantic validity and correctness of requirements. In order to detect inconsistencies that may lead to semantic errors or requirements that may have been omitted, we created some SPARQL queries below. The RDF data (graphs) uses the SPARQL query as a query language. We created the first SPARQL query to check which verbs (object properties) are connected to the class, *Money*, and have not been completed in the individual parts of boilerplates (*BoilerPlateParts*) as shown in Listing 1. The second SPARQL query, in Listing 2, checks which requirements we omitted to enter in category basic boilerplate <*subject*> <*verb*> <*object*>. In Listing 3, the SPARQL query checks if we have completed all the objects of the object property *sends*.

Listing 1. SPARQL query 1

```
SELECT DISTINCT  ?p
   WHERE {
      ?p rdf:type/rdfs:subClassOf* owl:ObjectProperty .

      ?p rdfs:domain | rdfs:domain/owl:unionOf/rdf:rest*/rdf:first  :Money  .
      FILTER NOT EXISTS {
         ?r rdf:type/rdfs:subClassOf* :BoilerPlateParts .
         ?r :verb ?p.
         ?r :subject :Money .

       }
}
```

Listing 2. SPARQL query 2

```
SELECT DISTINCT ?c ?p ?X
   WHERE {
      ?p rdf:type/rdfs:subClassOf* owl:ObjectProperty .
      ?c rdf:type owl:Class .
       ?p (rdfs:domain | rdfs:domain/owl:unionOf/rdf:rest*/rdf:first ) ?c   .
      ?X rdf:type owl:Class .
       ?p (rdfs:range | rdfs:range/owl:unionOf/rdf:rest*/rdf:first ) ?X   .
      FILTER NOT EXISTS {
         ?r rdf:type/rdfs:subClassOf* :Requirement .
         ?r :verb ?p.
         ?r :subject ?c .
         ?r :object ?X .
       }
}
```

Listing 3. SPARQL query 3

```
SELECT  DISTINCT ?c
  WHERE {
     ?c rdf:type owl:Class .
    :sends rdfs:range/owl:unionOf/rdf:rest*/rdf:first  ?c .
}
```

4.5 Summary of our Methodology

The classification of boilerplates that was developed by [38] includes: a) the requirements template and their legal obligation, b) requirements template without conditions, and c) the complete requirements template with conditions. The classification of the EARS boilerplates that were developed by [31], [32] includes: a) requirements template without conditions, and b) the complete requirements template with conditions. The difference between the two is that the former also considers the case where the verb has no object. The classification, we created based on requirements template without conditions, the requirements template with conditions which are distinguished to logical temporal conditionals, but boilerplates based on the linguistic nature of RDF triples that is similar to the syntax of the boilerplates. Table 10 summarizes all categories of our boilerplates.

Table 10. Categories of Boilerplates

No.	Boilerplate	Description
1.	*Basic boilerplate:* *<subject> <verb> <object>* *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]* *Extended basic boilerplate with objects as properties of classes,* *<subject> <verb> <object> of <entity>* *Extended basic boilerplate with subjects as properties of classes,* *<subject> of <entity> <verb> <object>* *Extended basic boilerplate with subjects as properties of classes and objects as properties of classes* *<subject> of <entity> <verb> <object> of <entity>* *Extended basic boilerplate with interaction and many objects* *<subject> <verb> <object> + From <entity>* *<subject> <verb> <object> + To <entity* *Extended basic boilerplate with timed operations* *<subject> <verb> <object> for <numerical-comparison-operator> <number> <TimeUnit>* *Extended basic boilerplate with interaction and many object as properties of classes* *<subject> <verb> <object> of <entity> + To <entity>* *<subject> <verb> <object> of <entity> + From <entity>* *or* *<subject> <sends> <object> of <entity> + To <entity>* *<subject> <receives> <object> of <entity> + From <entity>*	This boilerplate is called basic boilerplate. [Details] can also be applied anywhere. [Details] are explanatory comments.

No.	Boilerplate	Description
	<subject> <waits> <object> of <entity> + From <entity>	
2.	*Basic logical condition boilerplate:* *if basic boilerplate or extended basic boilerplate + then basic boilerplate or extended basic boilerplate +* *Extended* logical condition *boilerplate:* *if basic boilerplate or extended basic boilerplate + then basic boilerplate or extended basic boilerplate +* *else* *basic boilerplate or extended basic boilerplate +* Extended logical condition boilerplate *Nested if:* *if basic boilerplate or extended basic boilerplate + then* *if basic boilerplate or extended basic boilerplate + then* *basic boilerplate or extended 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 boilerplates
3.	*Temporal Condition boilerplate(After or When):* *After basic boilerplate or extended basic boilerplate +, basic boilerplate or extended basic boilerplate+* *When basic boilerplate or extended basic boilerplate+, basic boilerplate or extended basic boilerplate+*	Extends the basic template with temporal connectives to create sentences that involve temporal relations between entities. This boilerplate is called temporal boilerplate.

5. Evaluation

In the previous part of the paper, we have presented a minimal set of boilerplates and an ontology we created for engineering software for an ATM. The minimal set of boilerplates maintains flexibility on the one hand and generality on the other. The methodology for creating boilerplates was based on exploiting the natural language of boilerplate and we mapped them to RDF triplets, that also have similar syntax. In this section we present an experiment we have conducted in order to evaluate our proposed methodology. The requirements engineer can use the ontology and the minimal set of boilerplates without needing to learn anything new to make use the above combination, namely ontology and boilerplates.

The requirements engineer in our experiment uses boilerplates as follows: they select the appropriate boilerplate from the available list according to the requirement in natural language. Natural language requirements were elicited earlier in the process of requirements' specification. The available list of boilerplates includes the template of a basic boilerplate, the extended template of a basic boilerplate and complex boilerplate. The template of the complex boilerplate can contain logical and temporal constraints. The engineer can select any of the above boilerplates according to the requirement description. Then the engineer should fill the values of the boilerplate attributes selecting terms from the proposed ontology. The experiment we conducted to evaluate the proposed method is an observational case study. For the design of the experiment, we have been inspired from Runeson et al. [41].

5.1 Research Questions

To evaluate the efficiency and effectiveness of the proposed method, that is the use of the boilerplate and ontology for engineering requirements, we proposed and presented above, we have conducted an experiment concerning engineering a software for an ATM. The research questions to be answered by conducting the experiment are the following: the first question refers to whether the proposed method increases the quality of the specified. The second question refers to the proposed methodology, namely to the combination of the

ontology with the boilerplates during the process of specification of requirements, whether it is easy to use or not.

In the first research question, we recorded the participants' answers when asked to create a requirement a) in natural language, b) by selecting the appropriate boilerplates from the available list that we created and, finally, c) by combining the ontology and the boilerplates. In addition, we have recorded the time for completing the above tasks, and we have also judged the quality-accuracy of the produced requirements. Regarding the second research question, we evaluated the difficulty of choosing the appropriate boilerplate among the ones in the selected set and the selection of appropriate values from the ontology for the attributes of the boilerplates. The proposed methodology, which includes the combination of boilerplate and the ontology, is also the content of the specific evaluation.

5.2 Description of the experiment

The proposed methodology was based on the ontology and a minimal set of boilerplates which were designed for the engineering software for an ATM. First, participants read a general description of the system. The experiment is carried out in three phases. First, they write requirements for certain specifications in natural language. Second, they formalize the corresponding requirements with the help of boilerplates. Thirdly, they formalize the requirements with the help of the ontology and the boilerplates, filling the values of the boilerplate attributes using terms from the ontology.

The two participants were given a brief description of the ATM system for the first phase. In the second phase, they were given the ATM description of the system, the syntax of requirement boilerplates, as well as examples of boilerplates. In the third phase, first the participants were familiarized with the use of the ontology in the Protégé tool. After the participants studying the above, the experiment started. No help was given to the participants during the experiment.

5.3 Data Collection

After the participants completed three phases of the experiment, in addition to quantitative data collected during the experiment, qualitative data were also collected, collected in order to answer the research questions. We gave a questionnaire in the form of a personal interview to the participants to collect the qualitative data in order in order to answer the research questions. The Likert scale was used in the questionnaire. Also, their answers were justified. The questionnaire is presented in Table 11. Finally, the participants were asked more information on defining requirements for the three phases. The evaluator did not record the interviews but kept notes. The person who created the ontology and the syntax of boilerplates, is also the evaluator.

Table 11. Questionnaire questions

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)

5.4 Data Analysis

Based on the qualitative and quantitative data we got from conducting the experiment, some conclusions can be drawn. The analysis for the qualitative data collected was done by comparing the questionnaire values of the participants. In the qualitative analysis we also took into account what the evaluator observed during the experiment. In terms of quantitative analysis, the evaluator aims to evaluate the proposed methodology. Specifically, the subject of the evaluation was the specification of requirements in the three phases. The evaluator noted the number of common points presented by the participants during specification of requirements. Also, the number of errors is noted that occurred during the experiment.

5.5 Results

In Table 12 we recorded the quantitative results from specifying requirements with the boilerplates and the ontology. Participants specified requirements and their difficulty in specifying requirements is shown in Table 12. In addition, the Table contains results related to the overall understanding of the method. Finally, it contains results for the difficulty of selecting the values for the attributes of the boilerplates using terms from ontology.

Table 12. Quantitative results

Question	Engineer1	Engineer2
Overall understanding proposed method	0	1
Boilerplate identification difficulty	0	0
Placeholder identification difficulty from ontology	1	0

The participants did not face any difficulty regarding the method, i.e. the combination of boilerplates and the ontology, and they found the method understandable and easy to use. Additionally, the ontology we created describes the ATM software as well as the syntax of the boilerplate is based on concepts and relationships related to the above software. The participants' familiarity with the above concepts and relationships was expected to facilitate the use of the method. Also, before the experiment the participants were given to read the description for the ATM software as well as the ontology was given to familiarize them with concepts and relationships for the specific software. Clarifications were still given for concepts such as metaclass and examples to become familiar with the ontology before using the method. Regarding the syntax of boilerplate, the participants stated that it is quite understandable and expressive. About the ontology, they reported that the explanations about the ontology's operation were understandable to help users capture the requirements with the help of the ontology and boilerplates. Finally, the participants reported that they do not need to learn anything new in order to define requirements with the above method.

The appropriate selection of boilerplates during the second phase of the experiment was not difficult for them nor was the selection of appropriate values for the attributes in third phase. The more they became familiar with the syntax of boilerplates and the operation of the ontology, the easier was to choose the appropriate templates and values for attributes. We note that the people who took part in the experiment had little experience in specifying requirements as well as the ATM domain. According to Table 13, Table 14 and Table 15, the participants needed more time for complex boilerplates. Also, time-consuming procedures include finding the appropriate instances and the cases where the syntax boilerplates had metaclass as objects. However, as they stated, finding the above and, generally, the terms of the ontology were easy, if one started from object properties, that is, relationships between

subjects and objects. For the validity of the evaluation, we mention the small sample and the homogeneity of the sample.

We report a couple of examples of requirements and how they were formulated for the three different tasks in order to show the superiority of boilerplates and ontology regarding the quality of the generated requirement. For the following requirement: *Receive response from bank (about authorization)*, the first user wrote: *the computer responds to the ATM*, for task 1. In task 2, the first user chose the following boilerplate *<subject> <verb> <object>*. As the user filled the values of attributes of the boilerplate, they realized that they must change the wording of the requirement, so they finally chose the following boilerplate *<subject> <verb> <object> + From <entity>*, and the final generated requirement was the following: *<the ATM> <gets> <a response> from <the computer>*. In task 3, the user searched in the ontology for the appropriate entities, resulting in the following requirement: *<ATM> receives <Response> From <BankComputer>*.

The second user, for the same requirement in task 1, wrote: *the computer checks the authentication to send a reply.* In task 2, the second user selected the following boilerplate: *After <subject> <verb> <object>, <subject> <verb> <object>*. As the user filled the values of attributes of the boilerplate, they realized that they must change the wording of the requirement and selected the boilerplate *<subject> <verb> <object> + From <entity>*, resulting in the following requirement *<the ATM> <gets> <a response> from <the computer>*. Finally, in task 3, the user searched in the ontology for the appropriate entities and the following requirement was formed: *<ATM> receives <Response> From <BankComputer>*.

Another example is the following requirement: *The authorization starts after a customer has entered his card in the ATM.*, the first user wrote: *the computer has started the authentication process when it inserts the card*, for task 1. In task 2, the first user chose the following boilerplate *After <subject> <verb> <object>, <subject> <verb> <object>*. As the user filled the values of attributes of the boilerplate, they realized that they must change the wording of the requirement, and the final generated requirement was the following: *After <The user> <inserts> <the card>, <the computer> <starts> <the authentication process>, <the ATM> <gets> <a response> from <the computer>*. In task 3, the user searched in the ontology for the appropriate entities, resulting in the following requirement: *After <Customer> <enters> <CashCard>, <Authorization> <is_in_state> <startState>*.

The second user, for the same requirement in task 1, wrote: *the customer inserts the card and the check starts.* In task 2, the second user selected the following boilerplate: *After <subject> <verb> <object>, <subject> <verb> <object>*. The user filled the values of attributes of the boilerplate, resulting in the following requirement *After <the customer> <inserts> <the card>, <the computer> <starts> <checking>*. Finally, in task 3, the user searched in the ontology for the appropriate entities, and the following requirement was formed: *After <Customer> <enters> <CashCard>, <Authorization> <is_in_state> <startState>*.

Therefore, regarding the first question which explores whether the proposed method increases the quality of the specified requirements, we observe that users have started with slightly differently formulated requirements, in natural language, then both selected the same boilerplates to formalize the requirements, but again using slightly different words for the entities involved, whereas combining boilerplates and the ontology, “forced” them to end up in exactly the same formalization for the requirement. This means that our method effectively helps users to formulate accurate requirements, something that can prove beneficial at the later stage of requirements’ verification.

Regarding the time to complete the tasks, these are reported in Table 13, Table 14, and Table 15. Notice that task 2 took less time to complete than the other two, because users had just to select an appropriate boilerplate from a reduced list of predefined boilerplates, as they thought fit, after executing the first task of writing down a requirement in natural language, therefore having been familiarized with the syntax of the requirement. Furthermore, we notice that task 3 took more time to complete, because the users had to search for appropriate entities in the ontology, beginning their search from the main verb of the requirement, which is usually modelled as an object property in the ontology. Since most of these properties have a union of many classes in their domain and range and these classes have many instances, it took some time for the users to browse through them and, finally, select the correct one. Of course, our method could highly benefit from a dedicated tool that would visualize these dependencies and guide the users in their explorations to select the appropriate entities. Even with these reported times for task 3, the accuracy of the developed requirements, as reported above, makes our methodology worthwhile, reducing the ambiguity caused by natural language. Finally, we notice that the completion times for complex boilerplates are longer than the basic or extended boilerplates as they involve more complex sentences and relationships among entities.

The evaluation of the proposed method, i.e. the easiness of using boilerplates and an ontology for specifying requirements constitutes the content of the second research question. During the experiment as well as the interview, the participants did not face any difficulties in using the proposed method, neither in the selection of boilerplates nor in the selection of values for the attribute of boilerplates. As they became familiar with the use of the ontology, the selection of appropriate values for attribute improved, especially in the cases of metaclass and instances.

Table 13. Time in minutes for requirements specification

time for initial specification (in hours)	**Engineer1**	**Engineer2**
<subject> <verb> <object>	00:09.90	00:05.55
<subject> <verb> <object> From <entity>	00:14.74	00:17.02
If <subject> <verb> <object> then <subject> <verb> <object>	00:22.68	00:12.47
After<subject> <verb> <object>, <subject> <verb> <object>	00:36.22	00:07.08
Complex boilerplate *After <subject> <verb> <object>, <subject> <verb> <object>, <subject> <verb> <object> of <entity>* *if <subject> <verb> <object> of <entity> then <subject> <verb> <object> <subject> <verb> <object>*	01:48.43	01:46.87

Table 14. Time in minutes for requirements specification using only boilerplates

time for initial specification (in hours)	**Engineer1**	**Engineer2**
<subject> <verb> <object>	00:05.13	00:03.29
<subject> <verb> <object> From<entity>	00:07.21	00:07.28
If <subject> <verb> <object> then <subject> <verb> <object>	00:06.25	00:03.74
After<subject> <verb> <object>, <subject> <verb> <object>	00:05.25	00:03.55
Complex boilerplate	00:06.57	00:08.42

After <subject> <verb> <object>, <subject> <verb> <object>, <subject> <verb> <object> of <entity> if <subject> <verb> <object> of <entity> then <subject> <verb> <object> <subject> <verb> <object>		

Table 15. Time in minutes for requirements specification using both boilerplates and the ontology

time for initial specification (in hours)	**Engineer1**	**Engineer2**
<subject> <verb> <object>	00:19.63	00:02.38
<subject> <verb> <object> From <entity>	00:27.96	00:14.27
If <subject> <verb> <object> then <subject> <verb> <object>	00:24.60	00:21.90
After<subject> <verb> <object>, <subject> <verb> <object>	01:06.27	00:28.44
Complex boilerplate After <subject> <verb> <object>, <subject> <verb> <object>, <subject> <verb> <object> of <entity> if <subject> <verb> <object> of <entity> then <subject> <verb> <object> <subject> <verb> <object>	04:35.55	02:59.19

6. Conclusions

In this paper we have presented a methodology for creating system requirements based on semantic boilerplates with a corresponding domain ontology. We have presented and discussed the creation and classification of boilerplates based on the linguistic nature of RDF. We were also inspired this classification by the most famous boilerplate schemes which are EARS and the ones of Pohl and Rupp. In this paper we have created a boilerplate classification consisting of a) requirements template without conditions, b) requirements template with conditions, which are further differentiated to logical and temporal conditionals. What makes our methodology unique is that the syntax of our boilerplates is based directly on the linguistic nature of RDF triples. The case study we have used to explicate our methodology was a software engineering example for an ATM.

As we have presented, the basic syntax of boilerplates is subject – verb - object. Also, the syntax of the RDF data model has a similar, linguistic nature, namely subject - predicate (verb phrase) - object. Thus, the syntax of boilerplates and RDF is similar. We took advantage of this similarity, mapping boilerplates to RDF triples. Also, notice boilerplates consist of fixed and attribute elements. To create a boilerplate, the engineer has to fill in the blanks of attributes. Usually, the completion of the attributes is performed either manually or with the help of an ontology. In this work, we took advantage of the linguistic nature of RDF, which is similar to the one of boilerplates. We mapped boilerplates to RDF by storing the whole boilerplate (fixed and attribute elements) in the ontology and not only the attribute. This constitutes the novel contribution of this work, namely the fact that we exploit the syntax of the boilerplate language, mapping entire boilerplates to RDF triples.

The combination of ontologies and boilerplates is a useful tool that the requirement engineer should use to seamlessly formulate the requirements and at the same time ensure the quality and correctness of the requirements. This combination provides many benefits and reduces the effort of the process of requirements specification. Also, it can be used as a good guide and a good starting point for the inexperienced requirement engineers. Problems, such as ambiguity, related to the use of natural language in requirement specification are reduced. Another advantage is that the boilerplates and the ontology are reusable and can be renewed. Finally, combining ontologies (which have formal semantics) with boilerplates allows to carry out requirements validation checks, such as [33], [42] requirements incompleteness, requirements inconsistency, system model deficiencies based on requirements. This validation is based on ontological and rule-based reasoning [26], [27].

As a future work we will develop an application that will allow the engineer to create requirements and with the help of NLP and the corresponding domain ontology, it will suggest the appropriate boilerplate. Our solution can resolve semantic errors in requirements engineering that mainly involve relating wrong types of entities via certain types of boilerplates. However, our method can be extended with rule-based (or query based) reasoning for checking more semantical errors in RE, such as consistency, completeness, etc., as demonstrated by related works at other domains [26], [27], [34].

References

[1]Antoniou, G., Groth, P., Van Harmelen, F., Hoekstra, H. (2011). *A semantic web primer*. 3rd edition. MIT press.

[2]Anuar, U., Ahmad, S., & Emran, N. A. (2015). A simplified systematic literature review: Improving Software Requirements Specification quality with boilerplates. In *2015 9th Malaysian Software Engineering Conference (MySEC)*, 99-105. IEEE. 10.1109/MySEC.2015.7475203

[3]Arora, C., Sabetzadeh, M., Briand, L., Zimmer, F., & Gnaga, R. (2013). 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*, 35-44. IEEE. 10.1109/ESEM.2013.13

[4]Arora, C., Sabetzadeh, M., Briand, L. C., & Zimmer, F. (2014). Requirement boilerplates: Transition from manually enforced to automatically verifiable natural language patterns. In *2014 IEEE 4th International Workshop on Requirements Patterns (RePa)*, 1-8. IEEE. 10.1109/RePa.2014.6894837

[5]Arora, C., Sabetzadeh, M., Briand, L., & Zimmer, F. (2015). Automated checking of conformance to requirements templates using natural language processing. *IEEE transactions on Software Engineering*, *41(10*), 944-968. 10.1109/TSE.2015.2428709

[6]Berry, D. M. (2000). From contract drafting to software specification: Linguistic sources of ambiguity-a handbook version 1.0.

[7]Böschen, M., Bogusch, R., Fraga, A., & Rudat, C. (2016). Bridging the Gap between Natural Language Requirements and Formal Specifications. In *REFSQ Workshops*. http://ceur-ws.org/Vol-1564/paper20.pdf

[8]Cabral, G., & Sampaio, A. (2008). Formal specification generation from requirement documents. *Electronic Notes in Theoretical Computer Science*, *195*, 171-188. https://doi.org/10.1016/j.entcs.2007.08.032

[9]Campanile, L., Biase, M. S. D., Marrone, S., Raimondo, M., & Verde, L. (2022). On the Evaluation of BDD Requirements with Text-based Metrics: The ETCS-L3 Case Study. *In Intelligent Decision Technologies* (pp. 561-571). Springer, Singapore.

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

[11]Daramola, O., Sindre, G., & Moser, T. (2012). 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. https://doi.org/10.1007/978-3-642-33618-8_28

[12]Daramola, O., Sindre, G., & Stalhane, T. (2012). Pattern-based security requirements specification using ontologies and boilerplates. In *2012 Second IEEE international workshop on requirements patterns (RePa)* (pp. 54-59). 10.1109/RePa.2012.6359973

[13]Do, Q. A., Bhowmik, T., & Bradshaw, G. L. (2020). Capturing creative requirements via requirements reuse: A machine learning-based approach. *Journal of Systems and Software,* 170, 110730. https://doi.org/10.1016/j.jss.2020.110730

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

[15]Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., & Panis, C. (2011). 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. 10.1109/DDECS.2011.5783092

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

[17]Fantechi, A., Gnesi, S., Ristori, G., Carenini, M., Vanocchi, M., & Moreschini, P. (1994). Assisting requirement formalization by means of natural language translation. *Formal Method in System Design, 4*(3), 243-263. https://doi.org/10.1007/BF01384048

[18]Fernandes, R., & Cowie, A. (2004). *Capturing informal requirements as formal models* (Doctoral dissertation, Deakin University).

[19]Flemström, D., Afzal, W., & Enoiu, E. P. (2022). Specification of Passive Test Cases Using an Improved T-EARS Language. In *International Conference on Software Quality* (pp. 63-83). Springer, Cham. https://doi.org/10.1007/978-3-031-04115-0_5

[20]Fuchs, N. E., & Schwitter, R. (1995). Specifying logic programs in controlled natural language. *arXiv preprint cmp-lg/9507009*. https://doi.org/10.48550/arXiv.cmp-lg/9507009

[21]Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International journal of human-computer studies*, 43(5-6), 907-928. https://doi.org/10.1006/ijhc.1995.1081

[22]Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2), 199-220. https://doi.org/10.1006/knac.1993.1008

[23]Haris, M. S., & Kurniawan, T. A. (2020). 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). https://doi.org/10.1145/3427423.3427450

[24]Hull, E., Jackson, K., & Dick, J. Requirements engineering. Springer Science & Business Media, 2010.

[25]Ibrahim, N., Kadir, W. M. W., & Deris, S. (2009, December). Propagating requirement change into software high level designs towards resilient software evolution. In *2009 16th Asia-Pacific Software Engineering Conference* (pp. 347-354). IEEE. 10.1109/APSEC.2009.55

[26]Kravari, K., Antoniou, C., & Bassiliades, N. (2020). Towards a Requirements Engineering Framework based on Semantics. In *24th Pan-Hellenic Conference on Informatics* (pp. 72-76). https://doi.org/10.1145/3437120.3437278

[27]Kravari, K., Antoniou, C., & Bassiliades, N. (2021). SENSE: A Flow-Down Semantics-Based Requirements Engineering Framework. *Algorithms*, *14*(10), 298. https://doi.org/10.3390/a14100298

[28]Konrad, S., & Cheng, B. H. (2005). Facilitating the construction of specification pattern-based properties. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, 329-338. IEEE. 10.1007/978-3-319-94135-6_9
[29]Mahmud, N., Seceleanu, C., & Ljungkrantz, O. (2015). ReSA: An ontology-based requirement specification language tailored to automotive systems. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 1-10. IEEE. 10.1109/SIES.2015.7185035
[30]Mahmud, N., Seceleanu, C., & Ljungkrantz, O. (2016). ReSA tool: Structured requirements specification and SAT-based consistency-checking. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 1737-1746. IEEE. DOI: 10.15439/2016F404
[31]Mavin, A., Wilkinson, P., Harwood, A., & Novak, M. (2009). Easy approach to requirements syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, 317-322. IEEE. 10.1109/RE.2009.9
[32]Mavin, A., & Wilkinson, P. (2010). Big ears (the return of" easy approach to requirements engineering"). In *2010 18th IEEE International Requirements Engineering Conference*, 277-282. IEEE. 10.1109/RE.2010.39
[33]Mokos, K., & Katsaros, P. (2020). A survey on the formalisation of system requirements and their validation. *Array*, 7, 100030. https://doi.org/10.1016/j.array.2020.100030
[34]Mokos, K., Nestoridis, T., Katsaros, P., & Bassiliades, N. (2022). Semantic Modeling and Analysis of Natural Language System Requirements. IEEE Access, 10, 84094-84119.
[35]Musen, M. A. (2015). The protégé project: a look back and a look forward. AI matters, 1(4), 4-12.
[36]Natalya F. Noy and Deborah L. McGuinness, "Ontology Development 101: A Guide to Creating Your First Ontology", Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, March 2001.
[37]Panthum, T., & Senivongse, T. (2021). Generating Functional Requirements Based on Classification of Mobile Application User Reviews. In *2021 IEEE/ACIS 19th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 15-20). IEEE. 10.1109/SERA51205.2021.9509277
[38]Pohl, K. & Rupp, C. (2011). Requirements Engineering Fundamentals, 1st ed., Rocky Nook.
[39]Post, A., Menzel, I., Hoenicke, J., & Podelski, A. (2012). Automotive behavioral requirements expressed in a specification pattern system: a case study at BOSCH. *Requirements Engineering*, *17*(1), 19-33. https://doi.org/10.1007/s00766-011-0145-9
[40]Rosadini, B., Ferrari, A., Gori, G., Fantechi, A., Gnesi, S., Trotta, I., & Bacherini, S. (2017, February). Using NLP to detect requirements defects: An industrial experience in the railway domain. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 344-360). Springer, Cham.
[41]Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). Case Study Research in Software Engineering: *Guidelines and Examples*, 1st ed. Hoboken, NJ, USA: Wiley
[42]Stachtiari, E., Mavridou, A., Katsaros, P., Bliudze, S., & Sifakis, J. (2018). Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*, 145, 52-78. https://doi.org/10.1016/j.jss.2018.07.053
[43]Sunindyo, W., Melik-Merkumians, M., Moser, T., & Biffl, S. (2011). Enforcing safety requirements for industrial automation systems at runtime position paper. In *2011 2nd International Workshop on Requirements at Runtime* (pp. 37-42). IEEE. 10.1109/ReRunTime.2011.6046246
[44]Wang, Z., Chen, C. H., Zheng, P., Li, X., & Khoo, L. P. (2019). A novel data-driven graph-based requirement elicitation framework in the smart product-service system context. *Advanced engineering informatics*, 42, 100983. https://doi.org/10.1016/j.aei.2019.100983
[45]Warnier, M., & Condamines, A. (2017). Improving Requirement Boilerplates Using Sequential Pattern Mining. Europhras Conference, November 2017, London. http://dx.doi.org/10.26615/978-2-9701095-2-5_013
[46]Wleringa, R., & Dubois, E. (1998). Integrating semi-formal and formal software specification techniques. *Information Systems*, *23*(3-4), 159-178. https://doi.org/10.1016/S0306-4379(98)00007-6

[47]Zaki-Ismail, A., Osama, M., Abdelrazek, M., Grundy, J., & Ibrahim, A. (2020). Rcm: *Requirement capturing model for automated requirements formalisation*. arXiv preprint arXiv:2009.14683. https://doi.org/10.48550/arXiv.2009.14683

[48]Zichler, K., & Helke, S. (2019). 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. http://ceur-ws.org/Vol-2308/ase2019paper03.pdf