algorithms

MDPI

Article

SENSE: A Flow-Down Semantics-Based Requirements Engineering Framework

Kalliopi Kravari, Christina Antoniou and Nick Bassiliades *

School of Informatics, Aristotle University of Thessaloniki, GR541 24 Thessaloniki, Greece;
kkravari@csd.auth.gr (K.K.); antoniouc@csd.auth.gr (C.A.)
* Correspondence: nbassili@csd.auth.gr; Tel.: +30-2310997913

Abstract: The processes involved in requirements engineering are some of the most, if not the most, important steps in systems development. The need for well-defined requirements remains a critical issue for the development of any system. Describing the structure and behavior of a system could be proven vague, leading to uncertainties, restrictions, or improper functioning of the system that would be hard to fix later. In this context, this article proposes SENSE, a framework based on standardized expressions of natural language with well-defined semantics, called boilerplates, that support a flow-down procedure for requirement management. This framework integrates sets of boilerplates and proposes the most appropriate of them, depending, among other considerations, on the type of requirement and the developing system, while providing validity and completeness verification checks using the minimum consistent set of formalities and languages. SENSE is a consistent and easily understood framework that allows engineers to use formal languages and semantics rather than the traditional natural languages and machine learning techniques, optimizing the requirement development. The main aim of SENSE is to provide a complete process of the production and standardization of the requirements by using semantics, ontologies, and appropriate NLP techniques. Furthermore, SENSE performs the necessary verifications by using SPARQL (SPIN) queries to support requirement management.

Keywords: boilerplates engineering; designing software; ontologies; requirements engineering; semantics; software management

check for updates

Citation: Kravari, K.; Antoniou, C.; Bassiliades, N. SENSE: A Flow-Down Semantics-Based Requirements Engineering Framework. *Algorithms* **2021**, *14*, 298. https://doi.org/10.3390/a14100298

Academic Editor: Haridimos Kondylakis

Received: 29 September 2021
Accepted: 12 October 2021
Published: 15 October 2021

1. Introduction

Requirements engineering is one of the most important, if not the most important, stages in systems development. Whether it is software or hardware systems or embedded systems, the need for well-defined requirements remains the same. Even to this day, most of the projects rely on natural language specifications with loose guidelines. Unfortunately, these loose descriptions express features or functions that should be firm and binding requirements of the system. In order to understand how important specifications are, it is enough to think that an incomplete or misinterpreted description can lead to an application restriction or even to improper functioning of the system that will be difficult to fix later [1,2]. To this end, in many cases the requirements text is accompanied by a free-form text in an attempt to help clarify the requirements.

In fact, most of the time when it comes to analyzing the requirements of a system to be developed, the computer engineer collects all of the available material, documents, notes, interviews, etc. Then, based on his own personal knowledge and perception, he/she tries to combine all of this most likely vague and conflicting information and draw the necessary conclusions to proceed with the implementation. However, in this way verifications and checks cannot be made, nor can all the possible relationships and concepts be found. As a result, the success or failure of a system development depends largely on its requirements and the way they were initially defined, understood, and interpreted. Hence, an approach that would support requirements formalization and provide a way of transforming the requirements through a systematic process into a formal (or even semi-formal) and consistent specification remains an open research issue [3–8]. However, this effort is difficult as the system requirements are not only divided into the functional, which states the functionality and the performance, and the non-functional, which states the quality and limitations, they also have many subcategories [9], such as those depicted in Figure 1.

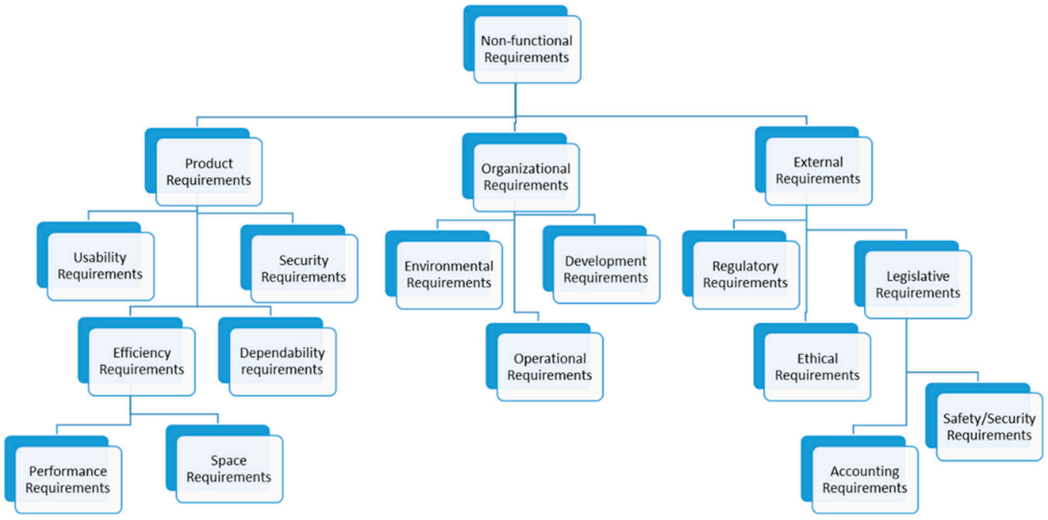


Figure 1. Non-Functional Requirement Categories, based on [10].

In this context, the main purpose of the proposed research is to support the development of systems using efficient and clear identification of their requirements. The approach combines semantics, ontologies, and natural language processing (NLP) techniques, with the aim of improving the current practice of using natural language to record the requirements of a system as this is one of the most important disadvantages of system development.

The proposed framework, called SENSE, contributes to the field of requirements engineering through:

(a) the establishment of integrated sets of boilerplates, namely standardized expressions of natural language with well-defined semantics;
(b) the proposal of the most appropriate standard expressions (boilerplates) for the systematization of a given requirement in natural language, based on the type of requirement (e.g., functional, performance, interfaces, design constraints, etc.), the type of system, the framework in which it is included, as well as an interactive process that takes into account the semantic distance between the requirements in the natural language and the standard expressions and proposes appropriate conversions;
(c) the proposal of systematic checks to verify the validity and completeness of the requirements, based on the boilerplate used, the type of requirement, the type of system, and the framework in which it is included.

Hence, SENSE proposes a complete (flow-down) process of production and the standardization of requirements (Requirements Specifications) that will include the minimum consistent set of formalities and languages to determine the requirements and perform the necessary verifications.

The rest of the article is organized as follows: Section 2 briefly discusses the notion of requirements, properties, and common taxonomies; Section 3 presents the SENSE proposed methodology; Section 4 discusses system testing; Section 5 presents a discussion of the findings; Section 6 presents related work; and Section 7 concludes, proposing future research directions.

2. Requirements, Properties, and Taxonomies

In SENSE, we study the so-called technical requirements which refer to the function, performance, interface, etc., of a system rather than the requirements related to the management of a project or business agreement which are out of the scope of this study. Furthermore, the main concern of the proposed approach is to propose an alternative to the requirements formalization problem. Although the requirements of a system can vary, depending on the type of system being developed and its specific needs, a requirement is a documented demand that software or hardware must be able to perform, or a specific constraint that it must have. In other words, a requirement could be a condition to achieve an objective or to satisfy a restriction or a contract. Hence, a requirement specifies capabilities, characteristics, and constraints that must be met [10].

As already mentioned, the requirements can be divided into the functional and the non-functional (Figure 1). The first category includes all the statements that describe the functionality that the system should provide, whereas the second category includes the statements that describe the quality of the system throughout its lifecycle. In this sense, the requirements correspond to the system and the software properties, and each property defines a set of expected behaviors or a set of constraints on the system implementation. Although it is difficult to associate the requirements with the specific properties as the requirements do not even correspond one-to-one with the properties, specifying constraints by using a state machine could lead from non-functional requirements to the properties.

In order to deal with these association issues and better understand the stages that are involved in requirements engineering, taxonomies can be used. A variety of published taxonomies can be found, such as that in [11], where non-functional properties are classified with respect to the different types of representation and the used parameters that determine the underlying measurement. Such taxonomies enable the design of a knowledge base that could support the formulation of a system's property set. Other taxonomies, such as that of IEEE, can guide us throughout the property elicitation, decomposition, and requirement coverage check. According to the research community and IEEE standards [12–14], the activities involved in requirements engineering involve four stages that are considered common and are always met [15–17]. These stages are the elicitation, the analysis, the specification, where the requirements are written down, and the validation, where the recorded requirements are checked for their consistency and their correspondence to the needs of the interested parties. These stages follow a specific chronological order, although in practice they often alternate with each other in the development of a system, especially if there were shortcomings during their first application. In practice, the requirements are used as inputs to the design stage. Yet, they also make a significant contribution to the verification process, as the checks should refer to the specific requirements.

3. SENSE Methodology

The aim of the SENSE framework is to provide an easy-to-use approach for systematizing a semantics-based specification of requirements/properties. For implementation purposes, it is based on a knowledge base that guides the engineers throughout the decomposition of the requirements and their coverage check.

3.1. SENSE Abstract Architecture

SENSE methodology comprises boilerplates, which are semi-complete requirements used as an input mechanism for capturing the underlying system-related semantics behind the requirements. Boilerplates are patterns that turn into requirements by restricting the syntax of sentences, using concepts from a system-specific ontology. In other words, these concepts are actually entities mapped onto the semantic model of the aforementioned ontology. An ontology is a logically defined vocabulary that contains concepts, their characteristics, and the relationships between them. It is thus possible to avoid indeterminate references in the specified requirements; to validate the specifications by ontology-based reasoning; and eventually to retrieve from the ontology the relevant information for the subsequent modeling activities. For the latter, the ontology is the means to capture implicit knowledge that has to be made explicit in order to ensure design correctness and consistency. Hence, a crucial factor of our approach is the ontologies presented below. As far as it concerns the holistic approach of the proposed SENSE methodology, it is developed in three main modules, as discussed in the following subsections, namely the development of the boilerplates (form and language standardization); the recommendation of appropriate boilerplates; and the systematic validity/completeness checks. Figure 2 depicts an abstract architecture of the system.

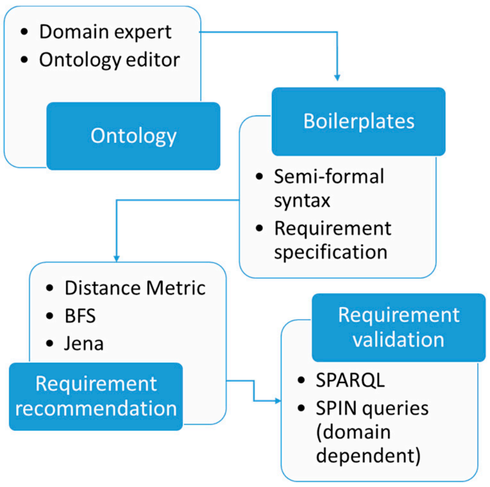


Figure 2. Abstract architecture of SENSE system.

In [18] the authors discuss the first steps towards this framework. There, only a first version of an ontology, called is U Ontology, was implemented along with the core version of the boilerplates language, which is now enriched with more clauses; whereas, the recommendation and validation functionalities were only referred to, as they were at their first steps. Since then, the whole system has been enriched, updated, and evaluated, forming the proposed SENSE framework, and two more ontologies have been developed. The isU Ontology was a limited-purpose ontology, used only for the purposes of the SENSE early development and testing. Although this ontology is still available, the system has been enriched with two rich ontologies, Shopy and ATM, as discussed below, that can be used by SENSE users for building the requirements of their own relevant systems. Furthermore, the language, as well as the functionality of the SENSE framework, is now fully developed, enriching the initial version, correcting the failures of the first implementation, and adding, among other aspects, validation functionality.

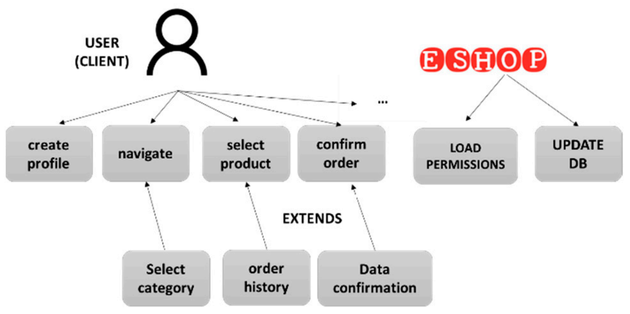
3.2. Ontologies

Given the importance of the ontologies, the first step of the proposed approach is to develop or adapt proper ontologies covering the system requirement specifications. Although SENSE is generic and can be used in a variety of cases, a proper domain ontology should be used each time in order to gasp the semantics and provide the necessary guidelines for a specific system. Officially, an ontology is a formal naming and definition of the concepts, their characteristics, and the relationships between them that exist in a particular domain. For example, a skyscraper (concept) is a kind of (relationship) building (concept) with floors (characteristic). Common components of ontologies include individuals, instances, classes, attributes, relations, functions, restrictions, rules, and axioms as well as events. Therefore, in this context, we have developed two ontologies, the first refers to an eshop software system called Shopy ontology, while the other refers to an ATM system, along with the necessary general (upper) ontology for the abstract description of the system. Discussing in detail the development of the ontologies is beyond the scope of this paper, yet they are briefly presented below.

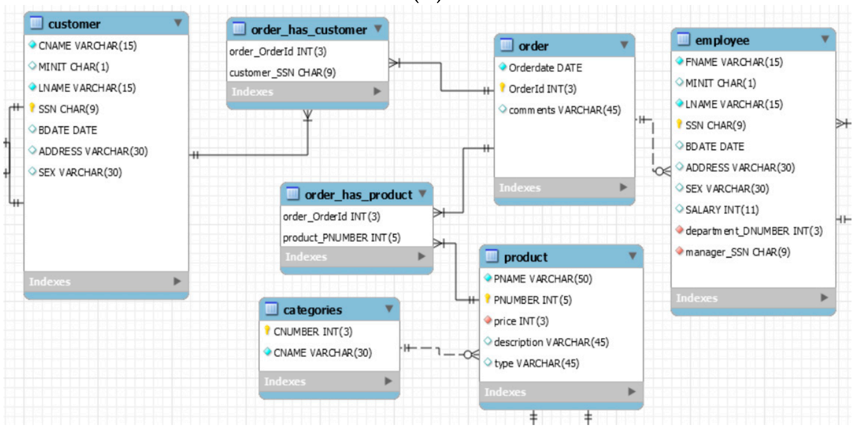
The Shopy ontology (Figure 3a–c) covers the requirements of an eshop, including company policies, stakeholders, and product management while it supports, among other aspects, sales and marketing. This system is able to handle the users, orders, and products of the company. The system users, staff, and clients can log in to the system and as soon as they are authenticated; they can, e.g., confirm a purchase or enter the payment information. For instance, Figure 3b depicts some of the core actions that such a system should support; it should allow users to create a profile (see Appendix A R1.6) and let them log in to the system (R1.7—"load permissions" in Figure) or select a product (R1.1). For the purposes of better understanding, the Appendix A presents part of the functionality and security requirements of the Shopy system.

```
<Class rdf:about="&shopy;ManageOptions">
            <rdfs:subClassOf rdf:resource="& shopy;ShopyFuncOptions"/>
</Class>
<Class rdf:about="&shopy;B2C">
            <rdfs:subClassOf rdf:resource="&shopy;CommerceTypes"/>
</Class>
<Class rdf:about="&shopy;SoftwareType">
             <rdfs:subClassOf rdf:resource="&shopy;LicenceTypes"/>
            <disjointWith rdf:resource="&shopy;CommercialType"/>
</Class> ….
```

(**a**)



(**b**)

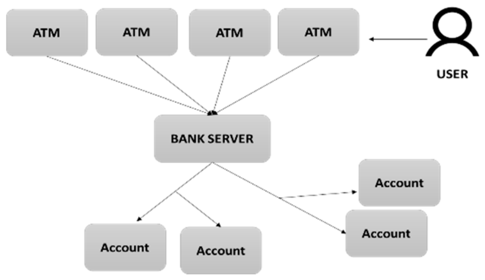


(**c**)

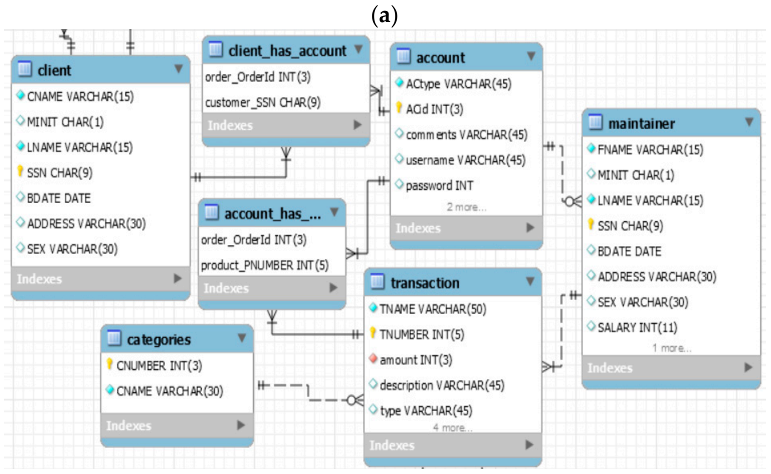
Figure 3. (**a**) Part of the Shopy ontology in RDF/XML; (**b**) part of functional requirements of Shopy system regarding a new client; (**c**) part of the Shopy ER diagram.

For implementation purposes, the TopBraid Composer, an editor (free edition) for the Resource Description Framework (RDF) Schema and OWL models, was used. This editor is fully compliant with the W3C standards, and it is considered appropriate for the ontologies and semantic applications. Furthermore, it supports, among others, the SPARQL Inference Notation (SPIN), which is used later in the SENSE framework.

The second ontology refers to an automated teller machine network, a scenario case used multiple times in academic courses [19]. The developed ATM ontology (Figure 4a,b) covers the requirements of an ATM network that supports server maintenance for the bank accounts as well as processing the transactions against them.



(**a**)



(**b**)

Figure 4. (**a**) Part of requirements of an ATM network regarding accounts; (**b**) part of the ATM ER diagram.

The system includes both the clients and the staff (maintainers) that, among others, are the only ones allowed to connect a new ATM to the network. The system accepts cards, authenticates the client (who can use any ATM in the network) and carry out the transaction if it is allowed. For instance, Figure 4a depicts the need of the ATM core system to support multiple ATM devices while each user is able to use any of these ATMs. Furthermore, each user may have more than one account. Hence, the bank server should handle all the users, regardless of which ATM they use, allowing them to log in if their credentials are proper (R1.3 Appendix A), while only one client may be served each time (R2.1). Part of the above functionality is presented in the Appendix A.

3.3. Form and Language Standardization

In order to address the ambiguity found in the natural-language written requirements, SENSE proceeds with systematizing the specification of the requirements by developing and establishing complete sets of standard expressions (boilerplates), using an appropriate artificial language. This language looks natural while allowing the combination of different boilerplates through a well-defined syntax and precisely defined semantics using reasoning procedures. In this context, natural language processing (NLP) techniques, such as the phrasal semantic parsing, are used both for better defining the boilerplates and for properly receiving the information embedded in them.

Practically, boilerplates allow the requirements to be recorded consistently, reducing, among other aspects, misspellings, poor grammar, and ambiguity, making it easier to understand, categorize, and identify the requirements. In addition, boilerplates act as an input mechanism to record not only the requirements but also the semantics that these requirements imply, e.g., who those involved are, or what the capabilities of the system should be [10,20]. Boilerplates are semi-complete and customizable in order to facilitate the user and to be able to suit the needs of different systems. The mechanism of boilerplates is, in essence, a fixed dictionary of concepts from a knowledge base, relevant to the field of hardware or software being developed, and a set of predefined structures. The basic building block of a boilerplate is a clause which expresses some aspect of the requirement. Each of such terms has a type, e.g., capacity, function, etc., which indicates the type of requirement being expressed. In addition, each term may have a type of target, indicating the general objective pursued, e.g., minimizing something, maximizing something, or exceeding a certain value, which is useful when we think about what exactly it is that we want to express. The clauses can be combined and put together to produce complete boilerplates, which will express multiple aspects of a specific requirement.

The semantics of the boilerplates language are matched with an appropriate ontology [21], such as those presented above, which is a particularly important aspect of the approach as it is the one that allows words to be drawn about the entities and their properties involved in the relationship described by the boilerplates. In any case, the ontology should focus on the concepts needed to define the requirements. Thus, the content of the boilerplates can, through reasoning, be examined for inconsistencies, omissions, etc. Practically, each boilerplate is defined as a sequence of attributes and fixed syntax elements, which facilitate the construction of a requirement. An example boilerplate could be: *IF <state>, <subject> shall <action>*. Obviously, the terms IF and shall are fixed components while the rest, enclosed in brackets, are properties that can take any value. This example consists of a prefix, *IF <state>*, which defines a condition, and a main clause, *<subject> shall <action>*, which defines a capability. For instance, requirement R19.2: "If user approves data insertion, the system updated DB" can be expressed using the aforementioned boilerplate as follows: *If <state> user approves data insertion </state>, <subject> the system </subject> shall <action> updates DB </action>*.

In fact, the language of boilerplates is a means of avoiding ambiguity in writing the requirements, in contrast, that is, to free writing, which exists in natural language. In such a language, certain requirements may be expressed through a number of alternatives, considering different provisions and combinations of clauses. Thus, the grammar (Tables 1 and 2) is context-free, containing both necessary and optional clauses, in the sense that some information types must always be written while the others are optional. It is obvious that everything that is required is explicitly stated. Actually, boilerplates consist of up to three different types of clauses, a *main* clause, which is mandatory, and two optional clauses, namely the *prefix* and the *suffix* clauses. Some requirements may be complex and require more than one clause to describe, hence the boilerplate may include multiple prefixes or suffixes separated by logic connectives (e.g., and, or). However, each requirement should have a single main clause for reasons of explicit reference.

Table 1. Boilerplate Grammar Notation.

Notation	Meaning
[. . .]	Optional
. . . | . . .	Or
*	>= 0 occurrences
+	>= 1 occurrences
< . . . >	boilerplate attribute

Table 2. Boilerplate Language Syntax.

<boilerplate> ::= <prefix><main><suffix>
<prefix> ::= <simple prefix> <logic connective> <prefix> | <simple prefix>
<suffix> ::= <simple suffix> <logic connective> <suffix> | <simple suffix>
<logic connective> ::= or | and | xor
<simple prefix> ::= <P_i>
<simple suffix> ::= <S_j>
<main> ::= <M_k>

As far as it concerns the main clause, the second objective of the present research, which was to recommend the appropriate boilerplates based on, among other aspects, the type of requirement, was also taken into account. To this end, specific types of main clauses were defined (Table 3) that support both the composition of the requirements, and their management. In this context, clauses that refer to a subject or an entity were defined among others. The clauses of the first case refer to the execution of an action, to an action-block, to a specific state, etc., while the clauses of the second case refer to when an entity designates another or when it is allowed to be executed.

Table 3. Boilerplate Main Clauses (M_k).

M1: <subject> shall [not] <action>
M2: <subject> shall [not] be <state>
M3: <subject> shall [not] be able to <action>
M4: <subject> shall [not] support <action>
M5: <subject> shall [not] execute <action-block>
M6: <subject> shall [not] allow <entity>
M7: <entity> shall [not] allow <entity> to <action>
M8: <subject> shall [not] handle <entity>
M9: <subject> shall [not] make available <entity>
M10: <entity> shall [not] be defined in <entity>

As for the prefixes, their main purpose is to correlate the main clause with preconditions referring to actions, states, or events. Thus, there are prefix clauses that specify when the main term should apply. Examples of this could be if/when an event occurs, when/while/if a situation exists, or when/while/if an action is performed (Table 4).

Table 4. Boilerplate Prefix Clauses (P_i).

P1: if | unless | while <state>
P2: if | unless <event>
P3: if | unless | while <action>

Finally, suffixes are used to configure or to calibrate the main clause by specifying additional information for the involved actions and entities. Thus, there are suffix clauses for events that set a start or an end limit with regard to the execution time or the frequency of an event (Table 5).

Table 5. Boilerplate Suffix Clauses (S_j).

S1: after | before <event>
S2: other than (<action>|<entity>)
S3: using <entity>
S4: without (<action>|<entity>)
S5: in the order (<entity>)
S6: at even intervals
S7: every | for a period of | within | for at least (time) <event>

This clear and unambiguous structure allows easy requirement specification as it has distinct sections that retain specific information. Of course, the effectiveness of a language in relation to its expressiveness and dynamics in eliminating ambiguity depends on the connectives used to relate sentences and their ontologically defined meaning. Connective words may be used to include additional entities in a specification or for determining time, order/sequence, purpose, consequence, comparison, contrast, and various types of conjunctions such as those for coordinating. A restricted set of connectives would limit the language's expressiveness, whereas a more extensive set of connectives could render it impossible to completely avoid ambiguity in the language syntax and semantics, which is an inherent problem in the design of any artificial natural-like language. Hence, a good practice is to assign multiple meanings to a connective word, which will be distinguished by the word's position within the sentence. Hence, SENSE uses the grammar rules in Table 6.

Table 6. Boilerplate Grammar Rules.

```
<subject> ::= system-function
<entity> ::= <simple entity> | <simple entity> <preposition> <entity>
<preposition> ::= to | in | on | from | with | without | between | among
<action> ::= <simple action> |<simple action><preposition><action>
<state> ::= <entity> mode
<event> ::= <entity> <action>
```

3.4. Boilerplates and Recommendations

Moving to the next step, SENSE proposes appropriate boilerplates depending on the type of requirement, the system and the framework in which it is included. In other words, the system provides non-binding recommendations, taking advantage of the information and relationships recorded in the ontology as well as those given by the user to the boilerplate. Hence, specialized recommendations are made. The approach combines two methods, namely controlled natural language (boilerplates) and phrasal semantic parsing (Apache Jena 22, ontology).

Jena is an open-source Java framework for building Semantic Web applications. It provides an API and extensive Java libraries that support programmers in extracting data from and writing to RDF graphs. Jena provides support for RDF, RDFS, RDFa, and OWL. Additionally, the extracted graphs that are represented as abstract models can be queried using SPARQL. Hence, Jena includes a rule-based inference engine to perform reasoning based on OWL and RDFS ontologies, and a variety of storage strategies to store RDF triples in memory or on disk. Of course, Jena is not a direct natural language processing (NLP) tool. This is because the notion of semantics of Semantic Web is not exactly the same as the notion of semantics used in natural-language processing. Yet, in this approach, Jena is used to develop a natural-language processor that emits data as RDF and then performs queries upon them. There are APIs and tools such as GATE 23 that are more oriented to NLP methodologies, and it is our intention to study them in the future, providing alternative functionality for the proposed framework.

At the present version of the SENSE framework, when the user writes a boilerplate (requirement in natural language), the system performs partial semantic analysis using Jena upon the knowledge base (ontology), proposing the closest semantics. Recommendations refer either to similar terms (general recommendations), e.g., "notebook" of Shopy ontology is connected to "laptop" via symmetric object property (isSynonymOf) or to the requirement categorization (specialized recommendations). At run-time, via the Jena reasoning mechanism, the requirement that is entered is dynamically categorized. The aim is to support engineers when writing clauses and to improve the quality of the boilerplate by combining the categorization of the requirements with the rest of the information.

For instance, the basic types of requirements are usually related to functional, performance, interface, design, and construction issues. Each of these categories has subcategories and clear specifications on how and what the requirements should be in the subsector. A typical example is the storage requirement, which is a basic requirement in the category of the functional requirements for the development of systems such as the ATM network. In this context, as soon as the user starts writing a functional clause, the framework extracts through reasoning the knowledge that storage space is required and informs the user (maintainer) about it with a message.

The approach is based on the following common semantic distance metric [21] that assesses the similarity between a given pair of terms by calculating the (shortest) distance between the nodes corresponding to these terms in the ontology hierarchy. The shorter the distance, the higher the similarity:

$$Dist(C_1, C_2) = \frac{2SPR}{D_1 + D_2 + 2SPR}\mathrm{e} \tag{1}$$

where D_1 and D_2 are, respectively, the shortest paths from C_1 and C_2 to C (their nearest common ancestor on the ontology hierarchy), and SPR is the shortest path from C to the root.

This metric was chosen as it is considered one of the most straightforward edge-counting methods when using ontologies where the ontology is faced as a graph that represents a connected word taxonomy. Hence, counting the edges between two terms can reveal the similarity between them. An example is depicted in Figure 5, where part of an ontology is depicted as a tree; each level of the tree (e.g., blue and red) reveals a similarity among the terms of this level. Furthermore, the BFS algorithm was chosen as it is able to find the shortest path between a starting term (node) and any other reachable node (second term). Part of the BFS algorithm, called BFS_RDF_Jena in SENSE, is presented below in pseudocode.

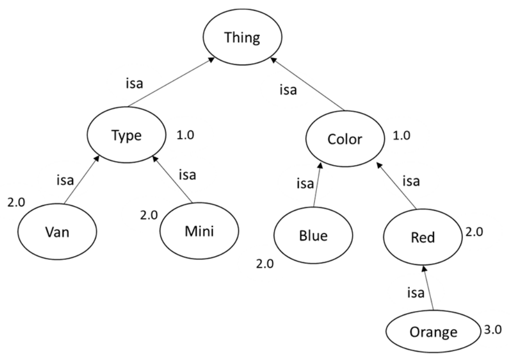


Figure 5. Part of an ontology represented as tree.

For implementation purposes, the OntTools class was used with the method Path findShortestPath (Model m, Resource start, RDFNode end, Filter onPath). It executes a breadth-first search, including a cycle check, to locate the shortest path from start to end, in which every triple on the path returns true to the onPath predicate.

Finally, given the ontology data, the methodology uses a SPARQL query to select all paths (Table 7) in order to study the recommendation module in depth.

Table 7. SPARQL Query Syntax.

```
select ?c1 ?c2 ?c3 where {
                values ?c1 { :a } ?c1 ^rdfs:subClassOf ?c2.
                OPTIONAL {
                              ?c2 ^rdfs:subClassOf ?c3}}
       order by ?c3 ?c2 ?c1
```

This SPARQL query returns the paths in the order that they would have been returned by a breadth-first algorithm. Yet, we adapted a direct breadth-first search, using Jena (Algorithm 1), to retrieve subclasses for the purposes of better understanding.

Algorithm 1. BFS_RDF_JENA.

```
BFS_RDF_Jena
    List <List<Resource>>
   BFS(model, <List<Resource>> queue, depth )
       Initialize List <List<Resource>> results
       while queue is not empty{
           set queue to List<Resource> path
          add results to path
          if (path size < depth) {
             Resource last = get path size minus 1
             Model list Statements ( RDFS.subClassOf, last )
             while (stmt has Next) {
                set List<Resource> extPath <> path
                get Subject as Resource (stmt next) and add to extPath
               queue offer extPath}
       return results
```

3.5. Validity/Completeness Checks

Finally, the framework provides necessary checks to ensure the validity and completeness of the requirements [8] and face an important challenge: to ensure that at the end we will end up with a set of requirements that are all covered by the properties, which can be enforced or verified and do not under-specify the system's behavior. For the first issue, it is necessary to identify any inconsistencies that could lead to semantic, structural, or behavioral contradictions, while for the second it is necessary to identify possible requirements that have been omitted when they should not have been. In fact, ontology, as a unit, is a static object-oriented model with constructional constraints. For example, a specific type of boilerplate can only be combined within the system with specific types of entities.

The framework deals with that by running appropriate SPARQL queries. SPARQL is an ontology query language. The questions, which can be formulated in this language, can identify issues regarding the requirements, the framework of the system, and the relationships that exist or are created. The advantage of the questions is that they need to be made once, although manually, by the expert, in this case a member of the project team, and then can be reused many times. For instance, at the ATM network each client that has an account must be at least 18 years old (Table 8).

Table 8. SPARQL Query and SPIN Constraint.

```
ASK WHERE {
    ?this client:age ?age.
    FILTER (?age < 18).
  }
```

```
[ a         sp:Ask ;
              rdfs:comment "must be at least 18 years old"^^xsd:string ;
                     sp:where ([ sp:object sp:_age ;
                                   sp:predicate my:age ;
                                   sp:subject spin:_this]
                                 [ a sp:Filter ; sp:expression
                                    [ sp:arg1 sp:_age ; sp:arg2 18 ; a sp:lt ] ]) ]
```

Another example, related to the ATM use case, checks if a user has the role of maintainer in order to access the bank server. The query (Table 9) checks if such permission is not completely covered by the requirements. The query checks if there are requirements related to Maintainer that should have the aforementioned permission (refers to the main clause of the M3 type (Table 3): <subject> shall [not] be able to <action>). Hence, there should exist a prefix clause of the type P3 (Table 4), for the same action, namely suspendAccount.

Table 9. SPARQL Query—ATM use case.

```
Select * WHERE {
        ?r a sense:User .
        ?r sense:hasRole ?x .
        ?x a sense:Maintainer.
         ?x sense:isRelatedToPermission ?Permission.
         NOT EXISTS { ?pr a sense:suspendAccount.
                      ?pr sense: isRelatedToPermission ?Permission.} .
```

Table 10 presents a query quite similar to the Shopy use case, related to a customer role. More specifically, it checks whether the user is a customer and, as a result, if he/she is allowed to select a product. The query checks if there are requirements related to Customer that should allow such an action (refers to main clause of the M7 type (Table 3): <entity> shall [not] allow <entity> to <action>, meaning that the system should allow the customer to act (select product). Hence, there should exist a prefix clause of the type P3 (Table 4) for the same action, namely selectProduct.

Table 10. SPARQL Query—Shopy use case.

```
Select * WHERE {
        ?r a sense:User .
        ?r sense:hasRole ?x .
        ?x a sense:Customer.
         ?x sense:isAllowedToAction ?AlAction.
         NOT EXISTS { ?pr a sense: selectProduct.
                      ?pr sense: isAllowedToAction ?AlAction.} .
```

Moreover, we define for each query a number of SPIN (SPARQL Inferencing Notation), a de-facto industry standard, and 19 constraints, which are SPARQL queries that are placed at appropriate classes of the ontology and, if evaluated positively, indicate a constraint violation from instances of the class. The aforementioned (ATM) query (Table 8) is transformed to the following SPIN Constraint (Table 11) while the subsequent query (Table 9) is transformed to the SPIN Constraint presented in Table 12.

Table 11. SPIN Constraint—ATM use case.

```
ASK WHERE {
   ? this sense:isRelatedToPermission ?Permission.
    NOT EXISTS {
         ?pr a sense: suspendAccount.
                  ?pr sense: isRelatedToPermission ?Permission .} .}
```

Table 12. SPIN Constraint—Shopy use case.

```
ASK WHERE {
   ? this sense: isAllowedToAction ?AlAction.
    NOT EXISTS {
         ?pr a sense: selectProduct.
                  ?pr sense: isAllowedToAction ?AlAction.} .}
```

Thus, finally with the use of SPIN, the rules can draw tacit relationships, fill in gaps, and fix the ontology and requirements by categorizing its instances while checking for all possible restrictions on instances of specific classes/types. SPIN allows SENSE to carry out ontology validation checks in three categories [22,23]:

1. Check incompleteness requirements.
2. Check for inconsistency requirements.

3. Check the system model for deficiencies based on the requirements.

In order to contribute to the quality of the requirement specifications the framework covers the following sub-cases:

- Non-initialization of abstract requirements: This validation check discovers abstract requirements that are not initialized for each instance of its main entity.
- Non-specification of abstract claims: This check detects abstract requirements that are not specified, in other words that there are no instances of the abstract entity. This validation check is, in essence, an indication of an incomplete requirement.
- Entities that are not related to demands: This check will return all entities that do not have a requirement. Thus, it can be used as a deficit check, identifying entities that are not yet associated with a requirement or entities that have ceased to be used as not required by a requirement, possibly due to new requirements or revisions.
- Conflicting requirements: This check will return pairs of requirements that may conflict with each other.
- Non-coverage of states of the system: This check will return the states that are not covered by a requirement.

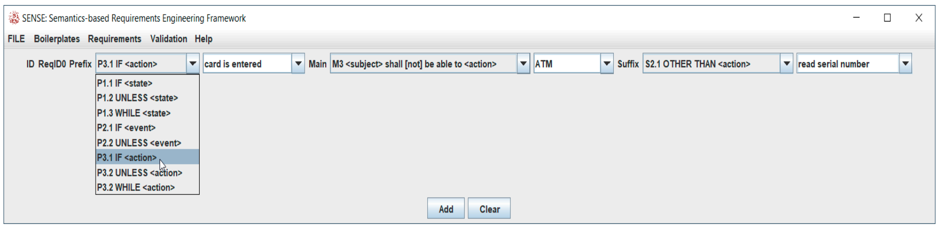
It has been pointed out that the control of requirements through ontology can only be static, i.e., it concerns the structural correctness and completeness of the system and not the dynamic/behavioral correctness. This can be checked only by converting the requirements into typically certain properties of the system, for the purpose of their algorithmic control (e.g., with model-checking techniques, etc.). The semantic-ontological approach of the present approach can help systematize such a perspective, as it standardizes both the requirements in terms of expressiveness and the system model in terms of permissible or non-permissible interfaces. Nevertheless, the expressive power of the ontologies is in the context of categorical first-order logic, while controlling the behavior of a system requires the use of temporal logics. Therefore, the conversion of the requirements into standard system properties and their systematic control are beyond the scope of this project.

4. Testing System

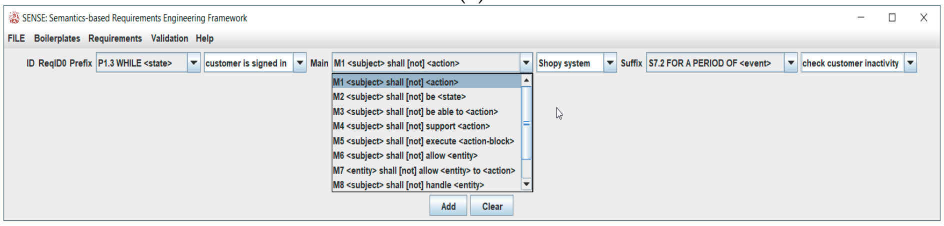
Finally, we conducted a statistic review in order to evaluate whether the approach can handle a user's faulty data insertion or not. To this end, a user used the SENSE GUI (Figure 6a,b) to state the requirement for both cases, ATM and Shopy. The experiment was conducted ten times. The user was a postdoc member of the research lab of the university and the experiment was conducted ten times as the user, after ten rounds, started to stabilize the understanding of the system and, as a result, her behavior.

Then, ontology validation checks in the three aforementioned categories were carried out. Figure 7 presents the statistical findings, which show that when the framework is used many times the outcomes are better, probably because the user better understands its functionality.

Finally, a percentage analysis of the sub-cases (Figure 8a,b) that the framework covers reveals not only the most common issues but also the added values of the approach in developing better requirement specification sets. For this analysis, the SPIN models (one for each use case, ATM and Shopy) that were used in the aforementioned experiment were used again in another experiment that was conducted ten times (5 times for each use case). The findings this time were recorded and categorized in five categories, namely non-initialization of abstract requirements, non-specification of abstract claims, entities that are not related to demands, conflicting requirements, and non-coverage of the states of the system. For instance, conflicting requirements mean that there were boilerplates that were assigned to the same subject and attributed different concepts. In this context, Figure 8a,b depicts the percentage appearance of the specific experiment. From the above experiments, the team concluded that when users become familiar with the framework they get better results in the requirements specification, while the framework itself, due to its (validation) functionality, supports a better quality of requirements.



(**a**)


SENSE: Semantics-based Requirements Engineering Framework
FILE Boilerplates Requirements Validation Help
ID ReqID0 Prefix
P1.3 WHILE <state>
customer is signed in
Main
M1 <subject> shall [not] <action>
Shopy system
Suffix
S7.2 FOR A PERIOD OF <event>
check customer inactivity
M1 <subject> shall [not] <action>
M2 <subject> shall [not] be <state>
M3 <subject> shall [not] be able to <action>
M4 <subject> shall [not] support <action>
M5 <subject> shall [not] execute <action-block>
M6 <subject> shall [not] allow <entity>
M7 <entity> shall [not] allow <entity> to <action>
M8 <subject> shall [not] handle <entity>
Add
Clear

(**b**)

Figure 6. (**a**) Part of SENSE GUI—Boilerplate (ATM use case); (**b**) part of SENSE GUI—Boilerplate (Shopy use case).

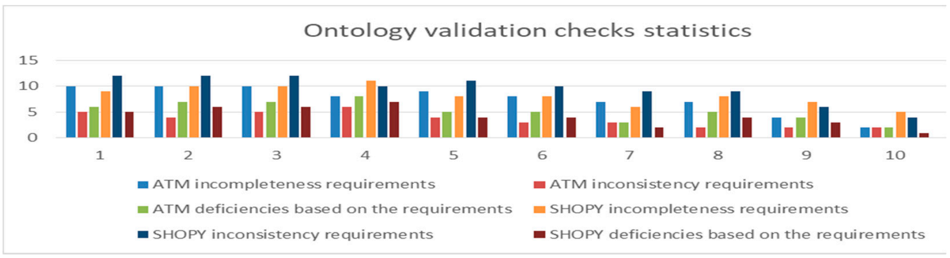
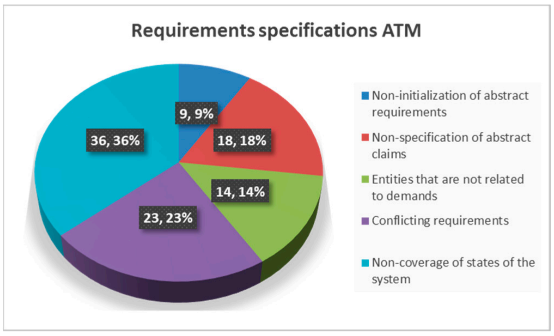
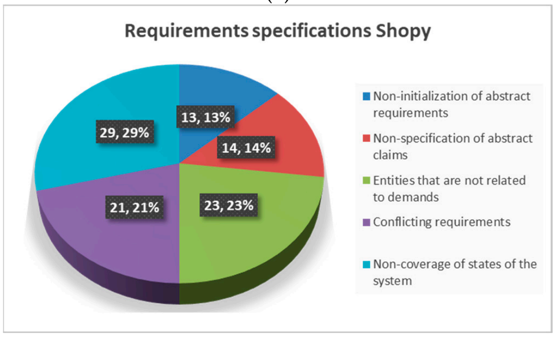


Figure 7. Validation statistics in ATM and Shopy use cases.



(a)



(**b**)

Figure 8. (**a**) Percentage analysis of the sub-cases in requirements specification for ATM use case; (**b**) percentage analysis of the sub-cases in requirements specification for Shopy use case.

5. Discussion

The proposed SENSE framework incorporates a minimum use of natural language with a formal method based on boilerplates. Hence, the approach provides a holistic framework that deals not only with functional and non-functional properties but also takes into account indirect, incomplete, or inconsistent cases that could affect the long-term behavior of a (developed) system. As a result, SENSE not only builds a set of requirements but provides an assistant that supports engineers in avoiding risks and uncertainties that could lead to high costs and late (software) delivery issues. It is strongly believed that tools, such as SENSE, can lead to sustainable engineering, which is actually the main aim of the proposed approach. SENSE can even be considered as a guideline tool that can assist software developers in the elicitation of sustainability requirements and the testing of software against these requirements if an appropriate domain ontology is used. This could be achieved as SENSE allows engineers to easily analyze their projects from the first design phase while paying attention to different sustainability dimensions, using an ontology and the validation checks provided by SENSE.

All-in-all, the SENSE approach enables the goals and objectives of the stakeholders to be revealed as boilerplates, and validity checks reveal the requirements in a solid way. Furthermore, everything can be documented, as boilerplates shift requirement development in a semi-formal procedure, which, although it is usually undervalued, is absolutely needed for both the implementation and the system operation monitoring purposes. Additionally, the approach supports steps that ensure clarity and transparency while avoiding assumptions (validity checks act as a safety valve), making the work of engineers easier and the system development more efficient. In this context, even if a requirement is not explicitly said or noticed by the experts and engineers, SENSE will reveal it. Another advantage is that engineers will be able to clarify what is actually an expert's wish rather than a goal. This will help them remain agile throughout the system development and better understand the necessary requirements. Finally, engineers can develop a valid set of requirements that can be used to get approval and explicit confirmation before starting to implement the system, minimizing misunderstanding and later modification costs.

Possible applications of SENSE could include cases where teams are building complex systems and products or when roles, processes, and requirement testing is required. SENSE, providing simplicity and customization, can be used by small businesses (such as start-ups) up to even larger companies that require better traceability and requirement management. In this context, SENSE can be used in an individual dimension when a single person is willing to develop a system for personal use or in a social dimension where conflicting interests should be taken into account by engineers in requirement defining. Of course, SENSE is mainly oriented to artificial systems, aiming at long-term maintenance and evolution. A non-extensible list of applications would include waste management and energy consumption applications as well as cloud-based, Internet of Things (IoT) and Semantic Web applications.

6. Related Work

Much of the research has focused on automation, with regard to translation and analysis, as an attempt to deal with natural languages and requirement variety. The common approach, in this context, is to translate a document written in a specific natural language to another formally defined language [10,24]. However, the ambiguity of natural languages makes translation difficult and the results of the process are in many cases doubtful. As a result, researchers acknowledge the need to shift from translation approaches to approaches that use standard (semi-) formal languages without the need for further translations, minimizing the use of natural language.

Additionally, there is another shift from the machine learning techniques that are used in the majority of the literature approaches to the procedures based on semantics. Semantics, namely the meaning of the requirements, allows the enhancement of accuracy and completeness, leading to a better understanding of hidden connections [8,10,25–27]. In the literature, however, there are even domain-specific approaches, such as in [28], that deal with a specific case, and such as with security requirements that propose visual notations in order to express requirements. In [28], the authors propose the use of business process management notation (BPMN), a graphical representation for business requirements to define business processes. In [29], a domain-independent tool that uses a UML model Generator provides diagrams and conceptual models. It actually uses NLP techniques and XML in order to provide a UML visualization. In [30], the approach is based on NLP in order to provide conceptual models using Extended Entity Relationship (EER) diagram notations. Finally, in [31] the authors propose an approach that transforms the requirements from natural language into semi-structured specifications. This attempt is based on Cerno, which is also a semantic annotation environment.

SENSE, on the other hand, provides an approach that allows engineers to develop, validate, and manage requirements in a well-defined way by using boilerplates and validity (SPIN) checks, minimizing the need for natural languages or machine learning techniques. Furthermore, once equipped with the appropriate ontology it can be used in any potential domain rather than in a few specific cases.

7. Conclusions

Requirements engineering is a core issue in systems development, determining either the success or the failure of a system. Despite the research efforts, there is still space for flow-down solutions regarding requirements specifications handling. In this context, this paper reported on a shift from translation to standard languages and methods, proposing a framework that minimized the use of natural language, using formal methods from the beginning. The approach uses semantics, ontologies, and the appropriate NLP techniques in order to provide a complete process of the production and standardization of requirements. The framework is called SENSE and it uses boilerplates, standardized natural language expressions with well-defined semantics to determine the requirements, including the minimum consistent set of formalities, while it performs necessary verifications using SPARQL (SPIN) queries.

As for future directions, we plan to develop more ontologies/system cases and test the framework. In this context, more verification statistical analysis will be conducted. On the other hand, we plan to use GATE [32], an NLP tool, in order to provide an alternative to the Jena-based [33] implementation. Of course, comparison between these two approaches will reveal weaknesses as well as strengths for the framework.

Author Contributions: Conceptualization, K.K. and N.B.; methodology, K.K. and N.B.; software, K.K. and N.B.; validation, K.K. and N.B.; formal analysis, K.K., C.A. and N.B.; investigation, K.K., C.A. and N.B; resources, K.K., C.A. and N.B; data curation, K.K., C.A. and N.B.; writing—original draft preparation, K.K.; writing—review and editing, N.B.; supervision, N.B.; project administration, N.B.; funding acquisition, K.K., C.A. and N.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research is funded in the context of the project «Requirements Engineering based on Semantics» (MIS 5047826) under the call for proposals «Support for researchers with an emphasis on young researchers-cycle B'» (EDULLL 103). The project is co-financed by Greece and the European Union (European Social Fund- ESF) by the Operational Programme Human Resources Development, Education and Lifelong Learning 2014-2020.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Part of Shopy Requirements Specification
Functionality
Ordered and Sell Products
R1.1 The system shall allow user to select a product
R1.2 The system shall display all the available product properties
R1.3 The system shall allow user (staff) to update the configuration of a product. R1.4 The system shall allow user (staff) to confirm the configuration
Product Categorizations
R1.5 The system shall display product categorization to the user (staff and client) *Customer Profile*
R1.6 The system shall allow user (customer) to create profile
R1.7 The system shall authenticate user (customer) credentials
R1.8 The system shall allow user (customer) to update the profile
Shopping Cart Facilities
R1.9 The system shall optionally allow user to print invoice
R1.10 The system shall provide shopping cart etc.
Security
Data Transfer
R2.1 The system shall automatically log out customer after a period of inactivity
Data Storage
R2.2 The system's databases shall be encrypted
R2.3 The system's servers shall only be accessible to authenticated administrators, etc.
Part of ATM Requirements Specification
Functionality
ATM Functionality
R1.1 The ATM has to check if the entered card is valid
R1.2 The ATM has to read the serial number of the card
R1.3 The ATM has to verify the bank code and password with the bank server
Client Options
R1.4 The client can abort a transaction
R1.5 The ATM also allows the client to clear off his/her bills
R1.6 A maintainer has the permission to suspend an account
Constraints
R2.1 The ATM must service at most one client at a time
R2.2 The maximum invalid pin entries is three, then the account is blocked
R2.3 The maximum amount of money a client can withdraw is 600€ per day
User Interface
R3.1 A screen will be provided for client to perform login
R3.2 A screen will be provided to display the other ATMs' location

References

1. Suhaib, M. Conflicts Identification among Stakeholders in Goal Oriented Requirements Engineering Process. *Int. J. Innov. Technol. Explor. Eng.* **2019**, *8*, 4926–4930. [CrossRef]
2. Khan, H.U.; Niazi, M.; El-Attar, M.; Ikram, N.; Khan, S.U.; Gill, A.Q. Empirical Investigation of Critical Requirements Engineering Practices for Global Software Development. *IEEE Access* **2021**, *9*, 93593–93613. [CrossRef]
3. Rajan, A.; Wahl, T. Requirements Engineering. In *CESAR—Cost-Efficient Methods and Processes for Safety-Relevant Embedded Systems*; Springer: Wien, Austria, 2013; pp. 69–143.

4. Hull, E.; Jackson, K.; Dick, J. Writing and Reviewing Requirements. In *Requirements Engineering*; Springer Science & Business Media: London, UK, 2010; pp. 77–91.
5. Liu, S.; Nakajima, S. Automatic Test Case and Test Oracle Generation based on Functional Scenarios in Formal Specifications for Conformance Testing. *IEEE Trans. Softw. Eng.* **2020**, *1*, 1. [CrossRef]
6. Mazo, R.; Jaramillo, C.A.; Vallejo, P.; Medina, J. Towards a new template for the specification of requirements in semi-structured natural language. *J. Softw. Eng. Res. Dev. Braz. Comput. Soc.* **2020**, *8*, 3. [CrossRef]
7. Barbosa, P.A.M.; Pinheiro, P.R.; De Vasconcelos Silveira, F.R. Towards the Verbal Decision Analysis Paradigm for Implementable Prioritization of Software Requirements. *Algorithms* **2018**, *11*, 176. [CrossRef]
8. Darlan, A.; Ibtehal, N. A Validation Study of a Requirements Engineering Artefact Model for Big Data Software Development Projects. In Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019), Prague, Czech Republic, 26–28 July 2019. [CrossRef]
9. Akbar, M.A.; Sang, J.; Khan, A.A.; Hussain, S. Investigation of the requirements change management challenges in the domain of global software development. *J. Softw. Evol. Proc.* **2019**, *31*, e2207. [CrossRef]
10. Sommerville, I. *Software Engineering*, 10th ed.; Pearson: London, UK, 2015; ISBN 13-9780133943030.
11. Pill, I.H. Requirements Engineering and Efficient Verification of PSL Properties. Ph.D. Thesis, Technische Universität Graz, Graz, Austria, 2008.
12. IEEE/ISO/IEC 21839-2019—ISO/IEC/IEEE International Standard—Systems and Software Engineering—System of Systems (SoS) Considerations in Life Cycle Stages of a System. 2019. Available online: https://standards.ieee.org/ (accessed on 16 June 2021).
13. IEEE 12207-2-2020—ISO/IEC/IEEE International Standard—Systems and Software Engineering—Software life Cycle Processes—Part 2: Relation and Mapping between ISO/IEC/IEEE 12207:2017 and ISO/IEC 12207:2008. 2020. Available online: https://standards.ieee.org/ (accessed on 16 June 2021).
14. IEEE/ISO/IEC 12207-2017—ISO/IEC/IEEE International Standard—Systems and Software Engineering—Software Life Cycle Processes. 2017. Available online: https://standards.ieee.org/ (accessed on 16 June 2021).
15. SWEBOK Version 3. Software Engineering Body of Knowledge. IEEE Computer Society. 2014. Available online: www.swebok.org (accessed on 14 August 2021).
16. Thayer, R.H.; Dorfman, M. *Software Requirements Engineering*, 2nd ed.; IEEE Computer Society Press: Piscataway Township, NJ, USA, 1997.
17. 29148-2018—ISO/IEC/IEEE International Standard—Systems and Software Engineering—Life Cycle Processes—Requirements Engineering. Available online: https://standards.ieee.org/ (accessed on 16 June 2021).
18. Kravari, K.; Antoniou, C.; Bassiliades, N. Towards a Requirements Engineering Framework based on Semantics. In *Proceedings of the 24th Pan-Hellenic Conference on Informatics (PCI 2020)*; Athens, Greece, 20–22 November 2020, Association for Computing Machinery: New York, NY, USA, 2020; pp. 72–76. [CrossRef]
19. Requirements Document for an Automated Teller Machine Network, Department of CSE, SDBCT, Indore, Software Engineering & Project Management Lab Manual. Available online: https://www.cs.toronto.edu/~{}sme/CSC340F/2005/assignments/inspections/atm.pdf (accessed on 16 June 2021).
20. Arora, C.; Sabetzadeh, M.; Briand, L.C.; Zimmer, F. Requirement boilerplates: Transition from manually-enforced to automatically-verifiable natural language patterns. In Proceedings of the 2014 IEEE 4th International Workshop on Requirements Patterns (RePa), Karlskrona, Sweden, 26 August 2014; pp. 1–8. [CrossRef]
21. Farfeleder, S.; Moser, T.; Krall, A.; Stålhane, T.; Zojer, H.; Panis, C. DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development. In Proceedings of the 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Cottbus, Germany, 13–15 April 2011; pp. 271–274. [CrossRef]
22. Mokos, K.; Katsaros, P. A survey on the formalisation of system requirements and their validation. *Array* **2020**, *7*, 100030. [CrossRef]
23. Stachtiari, E.; Mavridou, A.; Katsaros, P.; Bliudze, S.; Sifakis, J. Early validation of system requirements and design through correctness-by-construction. *J. Syst. Softw.* **2018**, *145*, 25–78. [CrossRef]
24. Anu, V.; Hu, W.; Carver, J.C.; Walia, G.S.; Bradshaw, G. Development of a human error taxonomy for software requirements: A systematic literature review. *Inf. Softw. Technol.* **2018**, *103*, 112–124. [CrossRef]
25. Perini, A.; Susi, A.; Avesani, P. A Machine Learning Approach to Software Requirements Prioritization. *IEEE Trans. Softw. Eng.* **2013**, *39*, 445–461. [CrossRef]
26. Clarke, E.M.; Emerson, E.A.; Sifakis, J. Model checking: Algorithmic verification and debugging. *Commun. ACM* **2009**, *52*, 74–84. [CrossRef]
27. Li, F.-L.; Horko, J.; Borgida, A.; Guizzardi, G.; Liu, L.; Mylopoulos, J. From stakeholder requirements to formal specifications through refinement. In *Requirements Engineering: Foundation for Software Quality*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9013, pp. 164–180.
28. Zareen, S.; Akram, A.; Ahmad Khan, S. Security Requirements Engineering Framework with BPMN 2.0.2 Extension Model for Development of Information Systems. *Appl. Sci.* **2020**, *10*, 4981. [CrossRef]
29. Deeptimahanti, D.K.; Babar, M.A. An Automated Tool for Generating UML Models from Natural Language Requirements. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, 16–20 November 2009; pp. 680–682. [CrossRef]

30. Vidya Sagar, V.B.R.; Abirami, S. Conceptual modeling of natural language functional requirements. *J. Syst. Softw.* **2014**, *88*, 25–41. [CrossRef]
31. Kiyavitskaya, N.; Zannone, N. Requirements model generation to support requirements elicitation: The Secure Tropos experience. *Autom. Softw. Eng.* **2008**, *15*, 2–149. [CrossRef]
32. van Erp, M.; Reynolds, C.; Maynard, D.; Starke, A.; Martín, R.I.; Andres, F.; Leite, M.C.A.; de Toledo, D.A.; Rivera, X.S.; Trattner, C.; et al. Using Natural Language Processing and Artificial Intelligence to explore the nutrition and sustainability of recipes and food. *Front. Artif. Intell.* **2021**, *3*, 115. [CrossRef]
33. Jena. The Apache Software Foundation. 2021. Available online: https://www.apache.org/ (accessed on 16 June 2021).