# From Language-Independent Requirements to Code Based on a Semantic Analysis

Mariem Mefteh
IT department, Mir@cl Laboratory
ENIS, Sfax University
Sfax, Tunisia.
Email: Mariem.Mefteh.Ch@gmail.com

Nadia Bouassida
IT department, Mir@cl Laboratory
ISIMS, Sfax University
Sfax, Tunisia.
Email: Nadia.Bouassida@isimsf.rnu.tn

Hanêne Ben-Abdallah
Faculty of computing and Information Technology, King Abdulaziz University,
Jeddah, KSA
Email: HBenAbdallah@kau.edu.sa

***Abstract*—This paper presents a new approach, which allows building Java codes from language-independent requirements. In other terms, it does not require any manual transformation of the requirements into the syntax of a specific programming language. To handle these challenges, our approach relies on a set of English-based mapping rules to generate a semantic representation of the input requirements. This semantic representation is used to produce the source code via existing code generators, such as Pegasus_f. Indeed, our approach extracts the Pegasus code from the semantic representation. This code is refined by eliminating the redundancy among the code elements' names thanks to the Term Frequency/Inverse Document Frequency (TF/IDF) method and the Density-based spatial clustering of applications with noise (DBSCAN) algorithm. Finally, Pegasus_f transforms automatically the resulting Pegasus code to Java. The proposed approach is implemented through the Code Recovery tool (CodeRec-tool), which accepts the English and the French languages in its actual version. The simplicity and the usefulness of our approach have been evaluated using measurements and based on experts' feedback.**

***Keywords*–Natural Language Processing; Semantics; Requirements; Naturalistic programming; Syntactic/Semantic grammar.**

## I. INTRODUCTION

All programming languages are progressing. Programmers, working within a company, are forced to update always their knowledge to recover this progress and to be able to use them. This leads to a significant waste of time in acquiring the programming languages' instructions and syntax. However, if programmers were able to express their program ideas in natural language, they would not have to transform them into programming language structures anymore. Programmers are obliged to transform their thoughts into the existing programming languages. Thus, it would be useful if we resort to the development of new ones, which are completely different from what exist. In fact, ideas are almost the same if we express them in several languages. It would be valid as long as the language, in which they are expressed, exists and is understood by people.

Referring to Knöll et al. [1], current programming techniques suffer from four main problems, namely: (i) the mental problem, which reflects the obligation of program ideas' adjustment to the conditions of a specific programming language (like restructuring them in the form of classes, methods and attributes in the object-oriented languages); (ii) the programming language problem, which reflects the mandatory implementation of the same program ideas and algorithms but in many ways depending on each programming language conditions; (iii) the natural language problem, i.e. the fact that people from different countries and working together are obliged to document and comment developed software in a well-known language, especially in English, which is less productive than using their mother tongue (they can make errors when using a non-native language if they could not use it correctly); (iv) the technical problem causing the waste of developers' time, spent for implementing and debugging the programs although ideas are unique. In fact, they still have to deal with minor issues like choosing the right character set and doing number conversions, instead of facing the really challenging tasks of programming: describing, modeling and enhancing the actual idea of a program [2]. These problems incur time loss and productivity decrease for software development companies.

To leverage the aforementioned problems, the project Pegasus [3] was elaborated as a new, naturalistic programming language. Naturalistic Programming means writing computer programs with the help of a natural language [1]. Pegasus accepts instructions written in a semi-natural language, and it produces the respective program accordingly. Besides Pegasus, several works were proposed to generate code from instructions written in a natural language, *cf.* [4]–[7]. The majority of these works is either semi-automated, or accepts inputs that are not written in a purely natural language. Similar to Pegasus, most of them require that the input instructions are in a particularly structured English format.

In this paper, we aim to address the gap between how we think and how we shall resort to operational details to explain the same ideas in several natural languages. To do so, we take advantage of the high stage of advancement achieved in Pegasus and the version Pegasus_f of code generator [8]. We extend this project with a new approach that lets Pegasus_f accept instructions written in any and purely natural language. This approach transforms the language-independent input requirements into a formal, semantic representation within the semantic model. This latter is based on a set of mappings, called mapping rules, that maintain the semantics among the input sentences. The semantic model was initially proposed in [9] [10]. In this paper, we enhance it with new features, useful for treating language-independent requirements. In addition, we apply the enhanced semantic model to different languages, like English and French, in order to show the applicability of our program generation approach, independently of the language used for the requirements specification.

To implement our approach, we created the CodeRec-tool, which automates all its steps. More specifically, we used the linguistic development environment NOOJ [11]. Indeed, we transformed the mapping rules into a NOOJ syntactic/semantic grammar for each supported natural language (English and French for the actual version of CodeRec-tool). The application of this grammar on input requirements generates their representation in the semantic model. This representation is then transformed into a Pegasus code that is finally converted automatically to a Java code using the Pegasus_f generator.

The remainder of this paper is organized as follows: In Section II, we overview existing approaches for information extraction from texts and source code generation from requirements. In Section III, we present our approach for synthesizing source code (in Java) from requirements written in different languages. Our approach is illustrated through the Library management case study [3]. Section IV overviews the implementation of our approach. In Section V, we present and discuss the results of an experimental evaluation of our approach. Finally, Section VI summarizes the paper and presents an overview of our future works.

## II. RELATED WORK

This section deals with the state of the art on (i) information extraction from texts, and (ii) source code generation from textual requirements.

### A. *Works for Information Extraction from Texts*

There is a large body of the literature that treats the problem of information extraction from texts. For instance, Glavas et al. [12] proposed the event graphs for structuring event based information from text; their system performs anchor (i.e., a word that conveys the core meaning of an event, e.g., "killed" or "bombing") extraction, argument (i.e., protagonists and circumstances of events, e.g., "agent", "time", "location") extraction, and relation extraction (i.e., temporal relation extraction and event coreference resolution). This system treats only the events, i.e., the situations that happen. Thus, we cannot rely on this work because extracting source code from natural language requirements necessitates exploiting various naturalistic entities, not only events; naturalistic types are types for programming, which are inspired by natural-language notions [3].

On the other hand, many works focused on the Frame Semantics and the FrameNet project [13]–[17]. The Framework Semantics is based on lexicons. A lexicon contains entries, which are composed of: (a) some conventional dictionary-type data, mainly for the sake of human readers, (b) FORMULAS that capture the morphosyntactic ways in which elements of the semantic frame can be realized within the phrases or sentences built up around the word, (c) links to semantically ANNOTATED EXAMPLE SENTENCES, which illustrate each of the potential realization patterns identified in the formula, and (d) links to the FRAME DATABASE and to other machine-readable resources such as WordNet and COMLEX [17]. The Framework Semantics assumes that the lexicon is made of a background knowledge, whose structure is represented by "frames"; The definition of a frame implies: (i) the discovery of participants, i.e., the frame elements and are defined as their unique semantic roles to the situation, (ii) the mandatory participants of a frame called core frame elements, and (iii) the optional participants, called peripheral frame elements [13] [15]. The model of frame semantics has attracted the attention of a number of linguists interested in the lexicon of a specialty field [18]. Besides, it was applied to the field of football (e.g., [19]), biomedicine (e.g., [20]), law (e.g., [21]) and environment (e.g., [18]).

Nobody can deny the importance of these works, in general, and the frame semantics approach, in particular. However, they are relevant for specific domains and frames, namely those which are already defined by them, unlike our approach, which is applicable regardless of the studied domain. Besides, the information of the type of a sentence (e.g., a definition, a statement, an assignment, etc.) cannot be determined by the frame semantics approach, although this fact is required for a relevant code extraction method. In this context, the semantic model is one of the best solutions for us while it gives adequate and precise information, relevant for the code derivation task thanks to the naturalistic entities that it relies on (see Section III-A).

### B. *Works for Source Code Generation from Textual Requirements*

Several works propose to generate source codes from requirements. For example, Franců et al. [6] propose a framework including a generator that produces an implementation in the form of methods. The major limitation of this work is the necessity of a manual processing to build a domain model, required by the generator. On the other hand, Smialek et al. [22] [7], Nowakowski et al. [23] and Kalnins et al. [24] propose approaches that transform the requirements, in particular behavior scenarios written in a semi-natural language, into UML models and the final Java code. These approaches are based on a special Requirements Specification Language (RSL) to express the use case scenarios of a system. The major limit of these approaches is that the use case scenarios must be pre-processed and written in a semi-natural language, according to the SVO grammar (i.e., in Subject+Verb+Object); this means that there is no use of "naturalistic" types like links, references, etc. For instance, a conditional sentence in RSL must be preceded by "=> cond:" in order to be treated.

On the other hand, Liu et al. [4] developed a tool, called Metaphor, that accepts program ideas written in English with the form of a story, and that generates the corresponding program template in Python; Metaphor mines nouns to program objects, verbs to functions and adjectives to properties. In their work, Cozzie et al. [25] proposed the Macho system; this uses a natural language parser that parses descriptions written in natural language into a simple program, by asking the programmer to provide one or more examples of correct input and output as unit tests. Özcan et al. [26] developed an intelligent natural language interface based on the Turkish language to create Java class skeleton and listing the class and its members; Turkish sentences are converted into instances of schemata representing classes and their members. These above works use simple mapping models that map nouns to objects and arguments, verbs to methods and adjectives to attributes. However, the semantics of a natural language sentence should not be exploited using only these types of mapping models. There are other facts that should be taken into account.

On the other hand, Gvero et al. [5] proposed a system that accepts free-form queries containing a mixture of English and Java, and it produces Java code expressions that take the query into account and respect syntax, types, and scoping rules of Java, as well as statistical usage patterns. This system focuses only on API-related queries, and not any type of instructions.

Overall, the majority of the existing approaches treat only requirements written in one single language (English in most cases). In contrast, ours accepts requirements written, theoretically, in any natural language. Moreover, it does not require a manual transformation of the requirements into the syntax of a specific language. These two merits rely on the concept of semantic model, which we introduce in Section III-A. Moreover, our approach relies on this model as a solution to hold (almost) all the semantics of the input requirements written in a purely natural language.

## III. OUR APPROACH FOR SOURCE CODE EXTRACTION FROM REQUIREMENTS

In this section, we describe our approach, which is composed of three main tasks: (i) extraction of the semantic model representation from input requirements written, theoretically, in any and in purely natural language, (ii) conversion of the resulting representation into a Pegasus code, and (iii) refinement and transformation of the Pegasus code to Java. Figure 1 shows the functional structure of our approach.

As illustrated in Figure 1, the proposed approach first applies the semantic model on the input language-independent requirements in order to extract their semantic representation as English-mapping rules (step 1 in Figure 1). Then, our approach applies some transformation rules to extract the Pegasus code corresponding to the extracted mapping rules (step 2 in Figure 1). Afterward, our approach refines the Pegasus code by eliminating the redundancy among the code elements' names (step 3 in Figure 1). The resulting Pegasus code is finally used as an input to the Pegasus_f generator to get the target Java code (step 4 in Figure 1). Note that our approach does not extract directly the Pegasus code from the input texts for two reasons: (i) Pegasus_f accepts Pegasus codes written only in the English language; our approach treats multilingual texts and extracts the corresponding Pegasus codes in English; (ii) Pegasus codes contain instructions written in a controlled natural language (i.e., following a specific syntax); our approach treats quite complex and ambiguous texts, from which it extracts relevant information useful for building the target Pegasus codes. These challenges are handled thanks to the semantic model, which can be considered as a transition model between the language-independent requirements and the Pegasus_f inputs (i.e., Pegasus codes).

In the remainder of this section, we detail the process followed by our approach, which we illustrate through the library management case study [3]. We choose this example because it contains ambiguous sentences' structures, proving the potential of our approach in extracting relevant information leading to good Java codes. The following texts (1 and 2) belong to our case study, where the first is written in the English language, and the second is in French.

1- "A library consists of several rooms containing shelves, on which stand books. A book has a three-letter key, which corresponds to the three initial letters of the surname of its first author. The books are ordered in the library by this key. If a visitor lends a book, then a new loan card is created. Besides, it is added to the card index box. Moreover, the book, as well as the name, the address and the telephone number of the visitor, are noted on the loan card. In addition, the actual date is put down; now the book is not lendable anymore. If a visitant returns a book, then the loan card belonging to the book and the visitor is thrown away; now the book is lendable again.".

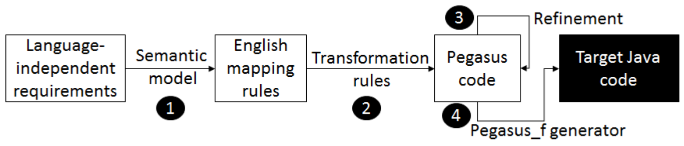

Figure 1. Functional structure of our approach

2- "Une bibliothèque se compose de plusieurs salles, contenant des étagères, sur lesquelles les livres se positionnent. Un livre a une clé de trois lettres, qui correspond aux trois lettres initiales du nom de famille de son premier auteur. Les livres sont ordonnés dans la bibliothèque par cette clé. Si un visiteur prête un livre, alors une nouvelle carte de prêt est créée. En outre, elle est ajoutée à la boîte d'index de la carte. De plus, le livre, ainsi que le nom, l'adresse et le numéro de téléphone du visiteur, sont notés sur la carte de prêt. Outre, la date actuelle est déposée; maintenant, le livre n'est plus prêtable. Si un visiteur retourne un livre, la carte de prêt appartenant au livre et au visiteur est rejetée; maintenant, le livre devient prêtable."

### A. *Requirements Representation within the Semantic Model*

In this section, we present the semantic model, which treats, in particular, the semantic nature of a sentence, as well as its constituents. In this context, Mitch Kapor states that "the critical thing in developing software is not the program, it's the design. It is translating understanding of user needs into something that can be realized as a computer program" [27]. In this sight, we proposed the semantic model as a first step towards representing formally raw ideas (i.e., following the way in which we think) independently of the used natural language and without resorting to operational details (like creating variables, defining their types, methods signatures, etc.). In fact, an idea would be represented always in the same notational way, no matter in which language it was originally expressed. For instance, the following sentences "A loan card is a card" (in English), "Une carte de prêt est une carte" (in French), "Eine Darlehenskarte ist eine Karte" (in German) and "Una carta di prestito è una carta" (in Italian) have equivalent meanings: the hierarchy (i.e., generalization/specialization) relationship between the objects "loan card" and "card". In this context, the semantic model analyzes the semantics among these sentences and represents them by one common representation. Originally [9] [10], the semantic model didn't handle all the characteristics of the natural language, so as to treat any type of sentences. In this paper, we explain in details this model's features. Furthermore, we accomplish it by new entities in order to be more useful for treating language-independent requirements.

The semantic model is based on many naturalistic entities, whose notation is inspired from [8], namely concepts, properties, actions, statements, sentences, references, compression, quantities and ordinalities. In addition, it supports the different types of loops, which are frequently used in natural language. Figure 2 shows the semantic model metamodel; it presents the semantic information that should be extracted from a language-independent instruction. More specifically, our approach relies on this metamodel to build the mapping rules; they represent a text, written in any natural language, in a formal and unique way. They are relevant to all natural languages. In other words, a sentence written in different languages will have the same representation as a mapping rule in English. In this section, we will present each mapping rule by using the EBNF notation [28] as follows: unquoted words denote a non-terminal symbol;

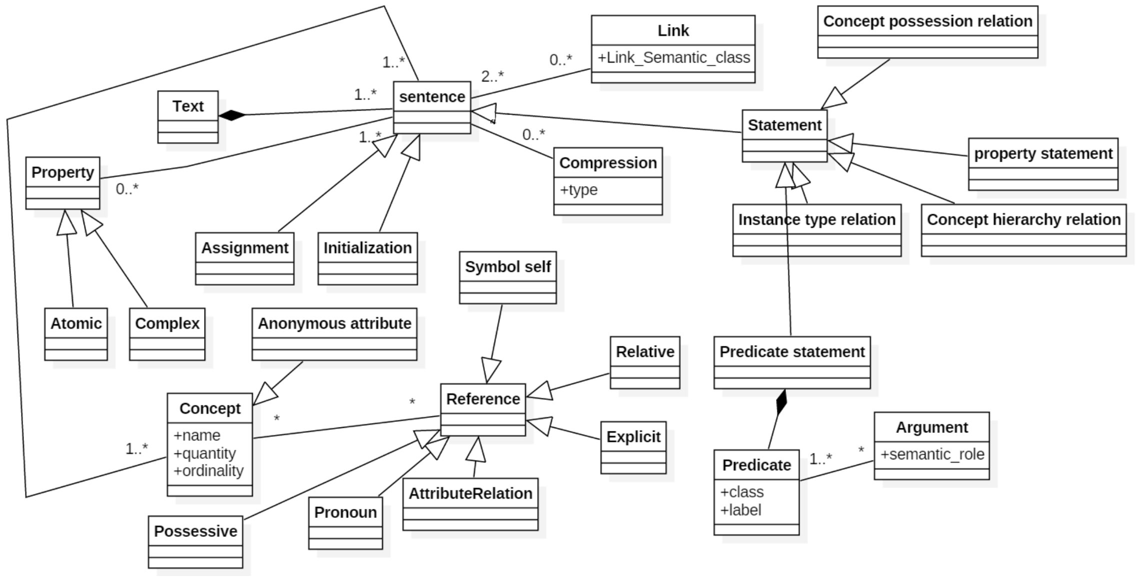

Figure 2. The semantic model metamodel

quoted words denote a terminal symbol, i.e., a symbol, which should be mentioned obligatory; the content of [] is optional; the content of {} denote symbols repeated zero or more times; the content of {}- denote symbols repeated one or more times; the character "=" denotes a definition; the semicolon denotes a rule terminator; the character "|" allows getting a choice from multiple options. In addition, some mapping rules contain the words "type" and "something". A "type" denotes the naturalistic entity. It can be a concept, a property, a quantity, statements preceded by a possessive pronoun, or a combination of these constituent [8]; and "something" stands for the most general type (it corresponds to the class "Object" in Java).

*1) Quantification and Ordinality:* Natural languages offer an elaborated system of quantification (e.g., "two books", "several books", etc.) over instances. Besides, ordinal numbers are used for counting, e.g., "first", "second", "third", etc. These two concepts are represented by the following mapping rule:

```
Quantity/Ordinality =
 "(quantity/ordinality," value ")";
```

*2) Naturalistic References:* Referencing is an integral part of natural languages. For example, we say "A book has a three-letter key, which corresponds to the three initial letters of the surname of its first author"; the words "which" and "its" are references, respectively, to "three-letter key" and "book". The semantic model accepts several references' types, such as:

- *Explicit reference:* it allows retrieving a subset of instances from a broader set of instances using the keyword "the" in English, e.g., "the card". It is represented in the semantic model by the following mapping rule:

```
Explicit reference =
 "(reference, explicit," type ")";
```

- *Symbol self reference:* it represents a symbol, which refers to a concept, e.g., "the button 'Loan'"; in this example "Loan" is a symbol that describes the concept "button". This reference is represented in the semantic model by the following mapping rule:

```
Symbol self reference =
 "(reference, symbol self," type, symbol ")";
```

- *Possessive pronoun reference:* it represents possessive pronouns in combination with a type, e.g., "its author". It is represented in the semantic model by the following mapping rule:

```
Possessive pronoun reference =
 "(reference, possessive pronoun," type ")";
```

- *Relative reference:* it is characterized by the use of relative pronouns like "which", "who", "whose", "with", etc. It is represented in the semantic model by the following mapping rule:

```
Relative reference = "(reference,
     relative," ["possessive," type] ")";
```

- *Attribute/relation reference:* it resolves expressions like "the type of the message". In fact, this type of expressions contains two other references. The first one placed before the word "of" is called a "filter reference". The second one, which is placed after the word "of", is called a "collection reference" [8]. The collection reference can be any reference; however the filter reference can be either an explicit or an ordinal reference. It can also consist simply of a concept. The attribute/relation reference is represented in the semantic model by the following mapping rule:

```
Attribute/relation reference = "(reference,
  attribute/relation," collection reference,
                        filter reference ")";
```

For example, considering the sentence "The books have a three-letter key, which corresponds to the three initial letters of the surname of its first author" from our case study; it is represented in the semantic model with the following mapping rule:

```
[...] (reference, attribute/relation,
 (reference, possessive pronoun,
  (author, (quantity/ordinality, first)),
 (reference, attribute/relation,
  (reference, explicit, surname), (letter,
   (quantity/ordinality, three), initial))))
```

*3) Concepts:* A concept is any “thing” from the real world, being abstract or concrete, e.g., “Book”, “Student”, “Amount”, etc. It is the homologue of an object in object-oriented programming languages. Concepts can contain or be contained in other concepts. This fact corresponds to the “concept possession relation”. Likewise, concepts can be sub-concepts of other ones. This fact represents the “concept hierarchy relation”. When we use a natural language, we do not create new instances explicitly like in existing programming languages. This process is done implicitly using some words like “take”, or by talking about non-existing things like “There are some books on the shelves”. The semantic model proposes the following mapping rule to represent the concept initialization (with its eventual properties) formally:

```
Concept initialization =
  "(initialization," concept ")";
```

In some cases, we may find a quoted string, which is not related to any concept. Thus, the semantic model treats it as a symbol with the following mapping rule:

```
Symbol definition= "(symbol," the_string ")";
```

Another type of concepts is the “anonymous concept”; it must be always contained into a concept and it describes different situations of this latter, using properties. For instance, let us consider the sentence “the type of a book can be science fiction, drama, action, romance, mystery or horror”; we note that the concept “type” is anonymous, belonging to the concept “book”, and supporting the values “science fiction”, “drama”, “action”, “romance”, “mystery” and “horror”, which serve to describe the concept “book”. We also note that an anonymous concept should contain neither other concepts, nor actions. Otherwise, it would be a simple concept. The anonymous concept is represented by the following mapping rule within the semantic model:

```
Anonymous concept definition = "(anonymous
  concept," concept "," {value}- ")";
```

For instance, considering the above example, it is represented in the semantic model with the following mapping rule:

```
(anonymous concept,
 (reference, attribute/relation,
  (book, (quantity/ordinality, abstract)),
  (reference, explicit, type)),
  (adjunctive, science fiction, drama, action,
   romance, mystery, horror))
```

where “adjunctive” is a type of compression (see section III-A6).

*4) Properties:* Properties describe concepts. There are several types of properties among which the type “simple” is the most used in the requirements. A simple property can be either the case or not, e.g., “short”, “lendable”, etc. It can be expressed by an adjective like “The shelve is empty”. A simple property is defined by assignment. It can be assigned in two ways: (i) directly, using the predicate “to be”, “can be”, “equal”, etc., which implies that there is a relationship between a concept and possibly several properties; or (ii) by initialization of a new instance. The semantic model represents these two mechanisms by the following mapping rules:

```
Property assignment = "(property assignment,"
 concept "," {property}- ")";
Property concept relation definition =
 "(property concept relation," concept ","
 {property}- ")";
```

For example, considering the sentences “Let the book be borrowable” (in English) and “Lassen Sie das Buch sein ausleihbar” (in German); they are represented by the same mapping rule, as follows:

```
(property assignment,
 (reference explicit, book), borrowable)
```

*5) Statements:* A statement is a declarative clause that is either true or false. We often use statements to express a relationship between different instances. The semantic model defines five types of statements, namely: Concept hierarchy relation and Concept possession relation statements, Predicate statement, Property statement and Instance type relation statement.

*a) Concept hierarchy and possession relation statements::* The two types of concept relations, “concept possession relation” and “concept hierarchy relation”, may appear both in concept definitions and in statements. These statements are respectively represented in the semantic model as the following mapping rules:

```
Concept possession relation = "(definition/
 statement, concept possession relation,
  (possessor," possessor_concept "),
  (possessed," possessed concept "))";
Concept hierarchy relation =
 "(definition/statement, concept hierarchy
  relation, (super-concept,"
  general_concept "), (sub-concept,"
  specialized_concept "))";
```

Note that the word “negation” is put when the statement (whatever is its type) is in the negative form. For instance, the clauses “A book has a three-letter key” (in English) and “Un livre a une clé de trois lettres” (in French) from our case study convey to one common representation within the semantic model, as follows:

```
(statement, concept possession relation,
 (possessor,
  (book, (quantity/ordinality, abstract))),
 (possessed, (key, (quantity/ordinality,
  abstract), three-letter)))
```

*b) Predicate statement::* This type of statements deals with predicates. A predicate refers to a verb. It belongs to a class (state or action). The difference between a state and a property is that this latter is unchangeable, whereas the state can change depending on the time, the location, etc. A predicate requires a number of arguments, which correspond to specific “semantic roles”. The semantic model treats, in particular, the following semantic roles (where some of them are introduced by the semantic model): (i) Agent: the entity that performs the action; (ii) Object: the entity that undergoes the action; (iii) Comparassant: designates the compared element as a part of a comparison; (iv) Comparator: designates the comparing element as a part of a comparison; (v) Possessor: something that has or contains someone/something; (vi) Possessed: something that is owned or in the disposal of someone/something; (vii) Sub-concept: a specialized concept in a hierarchical relationship; (viii) Super-concept: the generalized concept in a hierarchical relationship; (ix) Origin: the place from where an action is done; (x) Destination: the place towards which the action is directed; (xi) Location: the place or space of a predicate expressed by an action or a state; (xii) Time: indicates the date or the period when an action or a state is done; (xiii) Manner: describes the way of doing something. The predicate statements are represented in the semantic model with the following mapping rule:

```
Predicate statement = "(statement,
  (" predicate_class "," predicate "),"
  {"(" semantic_role "," parameter ")"}- ")";
```

For instance, considering the sentence: “Les livres sont ordonnés dans la bibliothèque par cette clé” (in French) from our case study; our approach generates the following English-mapping rule:

```
(statement, (action, order), (object,
  (reference, explicit, (book, multiple))),
 (location, (reference, explicit, library)),
 (manner, (reference, explicit, key)))
```

*c) Property statement: :* This type of statements focuses on properties. It is represented by the following mapping rule:

```
Property statement =
 "(statement, property," ["negation,"]
  ["comparative"|"superlative",] property_name
  ","{"(" semantic_role "," instance ")"}- ")";
```

For example, considering the sentences “A book has a three-letter key, which corresponds to the three initial letters of the surname of its first author” (in English) and “Un livre a une clé de trois lettres, qui correspond aux trois lettres initiales du nom de famille de son premier auteur” (in French) from our case study; our approach generates the same mapping rule for them:

```
[...]
(statement, property, (reference, relative),
 (reference, attribute/relation,
  (reference, possessive pronoun,
   (author, (quantity/ordinality, first)),
  (reference, attribute/relation,
   (reference, explicit, surname),
   (letter, (quantity/ordinality, three),
     initial)))))
```

*d) Instance type relation statement::* This type of statements takes interest in relationships between instances and their properties. It is represented in the semantic model with the following rule:

```
Instance type relation statement =
    "(statement, instance type relation,"
    ["negation,"] something "," type ")";
```

For example, considering the sentence: “If the type of the book is not drama...”; our approach generates the following mapping rule:

```
(condition, (statement, instance type,
 negation, (reference, attribute/relation,
 (reference, explicit, book),
 (reference, explicit, type)), drama)
```

*6) Compression:* Compression means grouping different syntactic structures together by some special words like “and”, “or”, etc. We distinguish four types of compression: copulative (using conjunctions like “and”, “added to”, “as well as”...), adjunctive (using conjunctions like “or”), contravalent (using conjunctions like “either.. or..”, “whether.. or..”) and exclusion (using conjunctions like “neither.. nor..”). The compression is represented in the semantic model following this rule:

```
Sentence = "(" ("copulative"|"adjunctive"|
           "contravalent"|"exclusion") ","
                          {something}- ")";
```

We will present an example of the compression in the following section.

*7) Sentences:* A sentence is composed of clauses linked by conjunctions (assembling links). A link belongs to a semantic class among the following ones: (i) Temporal: links two expressions in time; (ii) Condition: uses conditional conjunctions like “if”, “in case of”; (iii) Contrary: the opposite of condition using conjunctions like “if not”, “otherwise”, “else”; (iv) Final: expresses something happening as a result; (v) Cause: refers to a situation which is the cause of another situation; (vi) Illative: expresses something inferred from another statement or fact; (vii) Loop: represents five types of loops, namely: “for”, “while”, “do...while”, “foreach” and “switch”. The following mapping rule represents the link relation (excluding loops) within the semantic model:

```
Sentence =
 "(" link_semantic_class "," something ")";
```

For instance, the mapping rule corresponding to the sentence “If a visitant returns a book, then the loan card belonging to the book and the visitor is thrown away” is the following:

```
(condition, (statement, (action, return),
 (agent,
  (visitant, (quantity/ordinality, abstract))),
 (object,
  (book, (quantity/ordinality, abstract)))),
 (statement, (action, throw away),
  (object, (reference, explicit,
   (loan card,
    (statement, possession concept relation,
    (possessor, (copulative,
     (reference, explicit, book),
     (reference, explicit, visitor))),
    (possessed, (reference, relative))))))))
```

Concerning the loops links, the following mapping rules represent them:

```
Loop-do/while = "(loop," ("do"|"while") ","
  statement "," {something_result}- ")";

Loop-for = "(loop, for," counter_start_value
  "," counter_end_value "," step ","
  {something_result}- ")";

Loop-foreach = "(loop, foreach," concept ","
  statement "," {something_result}- ")";

Loop-switch = "(loop, switch," variable_name ","
  { value "," {something_result}- }- ")";
```

In summary, our approach works on requirements written in different languages, even the Asiatic ones, thanks to the semantic model. We refer the reader to our reference [29] for an example of our approach application on requirements, which are written in English, French, Spanish and Chinese languages.

## *B. Converting the Semantic Model Representation to a Pegasus Code*

The resulting mapping rules can be transformed into the input of existing code generators, such as Pegasus_f. This latter accepts requirements written in the Pegasus naturalistic programming language syntax, in English. Besides, it produces the corresponding Java code automatically. In this context, our approach transforms the mapping rules into the corresponding Pegasus code, based on some transformation rules. Due to space limitation, we will present only some of them (we suppose that the resulting Pegasus code is stored in the file "pegasus_code.peg"):

**Rule 1:** For each concept, $Conc$, involved within a mapping rule, a declaration of $Conc$ as a Pegasus concept is added to the file "pegasus_code.peg" following this syntax:

```
"concept: " Conc "{}"
```

We have to mention that our approach recognizes some keywords within the concepts' names (as well as the properties' and the actions' names). Therefore, it does not create the corresponding objects. For instance, our approach admits that each concept name, which contains at least one of the following words {"text", "string", "word", "letter", "paragraph", "line", "verse", "number", "integer", "float", "sum", "fraction", "numeral", "amount", "period", "menu", "menu-item", "button", "form", "option", etc.} do not correspond to a Pegasus concept. Indeed, Pegasus_f can recognize, automatically, the predefined Java types and GUI components, and thus, it does not require the creation of the corresponding Pegasus concepts.

**Rule 2:** For each property, $Prop$, in relation with a concept, $Conc$, a declaration of $Prop$ is added to the Pegasus concept $Conc$ in the file "pegasus_code.peg" according to the following syntax:

```
"concept: " Conc "{ property:" Prop "; }"
```

In fact, a Pegasus property declaration does not require the declaration of its type; this latter is deduced automatically by Pegasus_f.

**Rule 3:** Every concept possession relation within a mapping rule, involving a possessor concept, $c_{possessor}$, and a possessed concept, $c_{possessed}$, leads to the declaration of a Pegasus property $c_{possessed}$ within the concept $c_{possessor}$ following this syntax:

```
"concept:" c_possessor "{
  property:" c_possessed ";}"
```

**Rule 4:** Every concept hierarchy relation within a mapping rule, involving a sub concept, $c_{sub}$, and a super concept, $c_{super}$, leads to the declaration of the two Pegasus concepts $c_{sub}$ and $c_{super}$, where the first extends the second, by following this syntax:

```
"concept: " c_sub "extends" c_super "{}"
```

**Rule 5:** Each concept initialization within a mapping rule, involving a concept, $Conc$, and a property, $Prop$, is transformed to a Pegasus instruction according to the following syntax:

```
"let" Conc "be" Prop ";"
```

**Rule 6:** A copulative compression is transformed into a Pegasus syntax as follows:

- For each type, $typ$, involved within the compression, create a copy of the current mapping rule, in which the copulative compression clause is replaced by $typ$.

- Replace the current mapping rule by the new copies of mapping rules and treat them by applying the transformation rules adequate for them.

**Rule 7:** For each symbol self reference within a mapping rule, which involves a type corresponding to a concept, $Conc$, a Pegasus property called "label" is created within $Conc$. In fact, the involved symbol serves as a label to this concept.

We have to note that the concepts and the properties names convey to the standard notations, i.e., the units composing a noun are separated by putting the first letter of each term capitalized. After applying the transformation rules, we obtain the Pegasus code corresponding to the mapping rules of the input text. For example, our approach generates the following Pegasus code for our case study by applying the guidelines of the above rules on the extracted mapping rules:

```
concept: Library{ property:rooms;}
concept: Room{ property: shelves;}
concept: Shelve{}
concept: Book{
    property: key;
    property: author;
    property: isLendable;
    property: loanCard;
    action: to stand in (shelve){}
    (key) is ((three initial letters)
     of ((surname) of (first author)));}
concept: Author{
    property: isFirst;
    property: surname;}
concept: LoanCard{}
concept: Visitor{
    property: name;
    property: address;
    property: telephoneNumber;
    property: loanCard;
    action: to lend (book){}
    action: to return (book){}}
concept: Visitant{ action: to return (book){}}
[...]
(three-letter key) is ((three initial letters)
 of ((surname) of (first author)));
order (books) in (library) by (key)!
statement:{
    take (loan card)!
    add (loan card) to (card index box)!
    note (book) in (loan card)!
    note ((name) of (visitor)) in (loan card)!
    note ((address) of (visitor)) in
     (loan card)!
    note ((telephone number) of (visitor)) in
     (loan card)!
    put down (actual date);
    NOT((book) is lendable now);
 } : if (visitor) lend (book);
statement:{
    throw away (loan card)!
    (book) is lendable now;
 } : if (visitor) return (book);
```

Hence, this example shows that our approach is capable of generating a structured Pegasus code from quite complex and ambiguous input texts (containing sentences in the passive form with too much references), which are written in different languages.

### C. Pegasus Code Refinement

The input textual requirements may use synonymous words to describe the same concept. For instance, in our case study, the input text uses the words "visitor" and "visitant", which are semantically synonyms. They are however represented in the Pegasus code by two concepts, “Visitant” and “Visitor”, even though they are equivalent (see the generated Pegasus code in the previous section). To avoid such code redundancy and unify the names of Pegasus concepts, properties and actions belonging to the same concept, our approach uses an unsupervised classification of the names within the group of concepts names; this classification starts by getting the grammatical units from these names, by splitting these latter according to the capitalized letters used in them.

We choose the TF/IDF method [30], which relies on the calculation of the cosine similarity measure. This method is composed of queries and documents; our approach considers that a query consists of the units composing a Pegasus concept name, and a document is made up of the association of these latter, added to their synonyms extracted from WordNet [31]; WordNet is used only for the classification task, independently from the mapping rules synthesis process. TF/IDF begins by computing the weight of each grammatical unit, which composes a query $q_j$ and belongs to a document $d_i$. The weight of each unit is calculated thanks to the following equation:

$$w_{ij} = tf_{i,j} \times idf_{i,j} = tf_{i,j} \times \log(\frac{m}{D(i)}) \tag{1}$$

where: $w_{ij}$ is the weight of the grammatical unit $i$ in the document $j$; $tf_{i,j}$ is the frequency of the unit $i$ in the document $j$; $m$ is the total number of documents in the collection (i.e., the selected group of concepts, in this step); and $D(i)$ is the number of documents where the unit $i$ occurs. After that, our approach computes the cosine similarity, $Sim(d_i, q)$, between a document $d_i$ and a query $q$, using the following equation:

$$Sim(d_i, q) \approx \cos(\overrightarrow{d_i}, \overrightarrow{q}) = \frac{\sum_{t_j \in U} w_{ij} \times w_{qj}}{\sqrt{\sum_{t_j \in U} w_{qj}^2 \times \sum_{t_j \in U} w_{ij}^2}} \tag{2}$$

where: $w_{ij}$ is the weight of the grammatical unit in $d_i$; $w_{qj}$ is the weight of the unit $u_j$ in $q$; and $U$ is the set of grammatical units composing all the documents. Thus, our approach computes the cosine matrix (where the rows are the documents and the columns are the queries) according to equation 2. After performing this calculation, our approach applies the DBSCAN algorithm [32] on the cosine matrix, in order to group the concepts names into semantic classes. Then, it selects a name for each class. After that, it refines the generated Pegasus code by replacing the existing concepts' names by the selected semantic class names and merging the content of the initial similar concepts into the resulting concept names. These same steps are then applied for each group of actions and of properties that belong to the same Pegasus concept. To conclude, our approach generates a refined Pegasus code, which is then transformed automatically to Java using the Pegasus_f generator.

In the following section, we will present an implementation of our approach with the CodeRec-tool.

## IV. IMPLEMENTATION OF OUR APPROACH

To implement our approach, we developed a tool, named CodeRec-tool (Code Recovery tool), which allows generating a Pegasus code, as an input to the Pegasus_f generator, by starting from requirements written in different and in purely natural languages. Actually, this tool accepts the French and the English languages. However, it can be extended by integrating other languages.

This tool is composed of three modules: (i) the first module treats an input text and generates the corresponding mapping rules, which are then stored in a TXT file; (ii) the second module converts the mapping rules from the TXT file into a Pegasus code, which is stored in a PEG file, (iii) the third module refines the PEG file's content and uses it as an input to the Pegasus_f generator, which executes automatically the Pegasus code and generates the corresponding JAVA code.

Concerning the implementation of the mapping rules, we used the NOOJ environment [11] (our results are not completely dependent from the use of NOOJ; in fact, relying on this environment in the implementation of CodeRec-tool is just a choice.). NOOJ is a linguistic development environment that includes tools to create and maintain dictionaries, morphological and syntactic grammars. Dictionaries and grammars are applied to texts in order to locate morphological, lexical and syntactic patterns and tag simple and compound words [11]. Our main goal is to synthesize the different mapping rules that match each instruction of the input text to its formal representation within the semantic model in the English language. To this end, our tool does not follow a specific algorithm for the synthesis task, in contrast, it relies on grammars. Indeed, the mapping rules synthesis task is performed by developing a “syntactic/semantic grammar” that treats one specific language. In other words, in practice, we have to build a syntactic/semantic grammar for each natural language to be treated by our tool in order to generate the mapping rules of an input text written in that language.

Taking into account a text written in a language, $L$, different from English, our tool starts by exploiting this text by using the corresponding developed grammar, as well as the predefined NOOJ dictionary (available in [11]) appropriate to $L$. Then, our tool generates the corresponding mapping rules. However, these latter contain terms belonging to the language $L$. To solve the problem of the mapping rules translation to the English language, CodeRec-tool uses the Google Translate API [33] in order to transform the components of the generated mapping rules (i.e., the names of concepts, properties, states, actions, etc.) into English.

CodeRec-tool accepts the English and the French languages. To this end, we developed a NOOJ syntactic/semantic grammar for each language, including the syntactic and the semantic information of each language; these grammars allow to synthesize mapping rules in English, in cooperation with the predefined NOOJ dictionaries for the English and the French languages, as well as the Google Translate API. We refer the reader to our previous work [10] for a detailed example on the application of our NOOJ syntactic/semantic grammar on an input sentence.

Considering our case study, CodeRec-tool generates the corresponding mapping rules, which it stores in the file “Library_management_MRs.txt”. After that, it treats this file content, in order to extract the corresponding Pegasus code, which is stored in the file “Library_management_Pegasus.peg”. Indeed, CodeRec-tool implements the transformation rules from the semantic model representation to the pegasus syntax (see Section III-B).

The “Library_management_Pegasus.peg” file is then refined and used as an input to the Pegasus_f generator, which produces the corresponding Java code; it generates, in particular, the following extract of Java code:

```
public class Book{
 /*** Attributes declaration ***/
 private Key key;
 private Author author;
 private boolean isLendable;
 private LoanCard loanCard;
 /*** Constructors ***/
 public Book(){
  this.key=new Key();
  this.author=new Author();
  this.isLendable=false;
  this.loanCard=new LoanCard(); }
 public Book(Author author,
   boolean isLendable, LoanCard loanCard){
  this.author=author;
  this.key=this.author.surname.substring(0,3);
  this.isLendable=isLendable;
  this.loanCard=loanCard; }
 /*** Methods declaration ***/
 public void stand(Shelve shelve){}
 /*** Getters and setters ***/
 public Key getKey(){return this.key;}
 public void setKey(Key key){
  this.key=key;} [...] }
[...] }
```

We have to mention that our approach, and thus our tool, are able to generate, automatically, Java packages, as well. For example, let us consider the following sentence “The visitor selects the button ‘Lend”'; CodeRec-tool generates the following mapping rule and the corresponding Pegasus code:

```
Mapping rule
------------
(statement, (action, select),
 (agent, (reference, explicit, visitor)),
 (object, (reference, symbol self, button,
   "Lend")))
Pegasus code
------------
[...] (visitor) select (button "Lend");
```

Using this latter Pegasus code as an input to the Pegasus_f generator, CodeRec-tool generates, namely, the following Java code:

```
import java.awt.*;
import java.swing.*;
[...]
public class BookLending extends JFrame
               implements ActionListener{
 JButton button1=new JButton("Lend");
 button1.addActionListener(new ActionListener()
  { public void actionPerformed(ActionEvent e)
    {[...]}});
[...] }
```

## V. EVALUATION

To evaluate our approach and our tool, we adopted the process proposed by Wohlin et al. [34], which decomposes the evaluation into different parts, like goal, task, subjects, preparation, conduction and evaluation.

**Goal:** The overall objectives of our evaluation is to show the ability of our approach, and thus tool, in deriving useful Pegasus codes (implicitly good Java codes) from input requirements, written in purely and in several natural languages (English and French for its current version), and to examine the conformity degree between our Pegasus codes and those built by Pegasus experts. We rely on Pegasus experts in our evaluation, instead of Java programmers because the main outputs of our approach are Pegasus codes. Indeed, producing good Pegasus code leads, implicitly, to the generation of good Java codes.

**Subject and Preparation:** While a use case scenario contains useful description of a system behavior, from which we can deduce an important amount of source code, we decide to rely of the use case scenarios belonging to five different domains, and which are given to two Pegasus experts (two natural language processing PhD students from our laboratory, who are familiar to Pegasus programs), as follows:

- 16 use case scenarios belonging to a Health complaint application [35]; this latter allows citizens to report complaints (food, animal and special complaints) via internet.

- A well developed scenario of the use case “Withdraw cash” belonging to a banking system [36].

- 28 use case scenarios belonging to the Go-phone system [37]; this latter is based on a hypothetical context of the mobile phone company “Go-Phone” Inc and it has clone based Go-phone products, such as “S”, “L”, “Elegance”, “Com”...

- 9 use case scenarios belonging to a crisis management system [38]; this latter treats crisis, which can range from major to catastrophic affecting many segments of society.

- Use case scenario of the game of war cards [39], which involves two players where the one who has no more cards at the end of the game is the looser.

- 5 use case scenarios belonging to Emptio [40], which is a mobile phone application for selfservice shopping.

**Task:** We asked the Pegasus experts to give us the correct Pegasus codes corresponding to the adopted subjects.

**Conduction:** We compare the experts' Pegasus codes to the corresponding ones generated by our tool by using the precision and recall metrics in terms of Pegasus concepts ($pC$, $rC$), properties ($pP$, $rP$) and actions ($pA$, $rA$). For example these measures are calculated as follows for the concepts:

$$pC = \frac{number\ of\ true\ concepts}{number\ of\ found\ concepts} \times 100 \tag{3}$$

$$rC = \frac{number\ of\ true\ concepts}{number\ of\ real\ concepts} \times 100 \tag{4}$$

Moreover, we decide to use two other metrics taken from the standard ISO 25020, in order to measure:

- The completeness ($Com$) degree of our results according to the input requirements, which are also treated by the Pegasus experts; for example, it is measured, in terms of concepts, as follows:

$$Com_c = 1 - \frac{number\ of\ unfound\ pertinent\ concepts}{number\ of\ pertinent\ concepts} \times 100 \tag{5}$$

- The correctness ($Cor$) of our results according to the input requirements, which are also treated by the Pegasus experts; for instance, it is computed, in terms of concepts, as follows:

$$Cor_c = \frac{number\ concepts\ adequately\ implemented}{number\ of\ pertinent\ concepts} \times 100 \tag{6}$$

We have to mention that the correctness measure is only computed for the Pegasus concepts and the actions because it deals with their internal implementation. Besides, the number of concepts/actions adequately implemented means that the

TABLE I. EVALUATION

| Average | Precision | Recall | Complet. | Correct. |
|---|---|---|---|---|
| Concepts | 73,78% | 91,60% | 90,43% | 80,13% |
| Properties | 81,59% | 82,66% | 78,41% | - |
| Actions | 70,22% | 82,41% | 76,78% | 83,02% |

majority of their implementations are pertinent according to the experts' implementations.

Table I shows the resulting averages of the adopted measurements in terms of Pegasus concepts, properties and actions.

**Evaluation:** Table I shows the high average values of the adopted metrics, which exceed 73,78% in all cases. More specifically, the precision (respectively recall) values in terms of concepts range from 67,86% to 80% (respectively from 84,62% to 96%). Similarly, we note high average values of completeness and correctness, reaching respectively 90,43% and 83,02%. For example, in the case of the Go-phone system, we obtained the highest values of precision in terms of concepts (80%), with a completeness rate 91,67% and correctness of 87,5%. This fact means that our tool generates a good number of true positives (TP, i.e., the number of pertinent concepts generated by our tool), which equals 24, vs. a low number of false positives (FP, i.e., the number of none-pertinent concepts, not found by the experts and which are generated by our tool), which equals 6. Moreover, the recall value in terms of concepts for the Go-phone system equals 92,31%, reflecting that our tool generates the majority of pertinent concepts. The high values of completeness (91,67%) and correctness (87,5%) confirm this fact. Consequently, we deduce that the code generated by our tool is of a high quality and helps the developers in the programming task by saving their time, and thus money for the companies.

We have to mention that the false positives generated by our tool and decreasing the precision values in some cases (like the case of the Withdraw cash scenario, which equals 67,86% with 38 TPs and 18 FPs) are caused by the fact that our tool generates a Pegasus concept for each met concept within a mapping rule, except for some concepts whose names can be recognized by our tool and for which this latter does not produce a corresponding Pegasus concept (see Rule 1 in Section III-B). For instance, concerning the Emptio application, our tool generates, in particular, the concept "URL". In contrast, the Pegasus experts realize that this concept corresponds to a simple string and puts it as a Pegasus property within the Pegasus concept "Application". This fact is due to the full automation of our approach. However, we believe that the extra-generated concepts do not really matter because they will be removed by the developer, later. Indeed, the completeness (82,81%) and the correctness (81,82%) in terms of concepts for the Emptio application confirm the utility of the Pegasus concepts generated by our tool.

On the other hand, the precision values in terms of properties range from 75% to 96%. For example, in the case of the Health complaint system, our tool generates 42 TPs, vs. 12 FPs; in fact, the use case scenarios of this system treat each "type" of a "query" on its own. Thus, our tool generates four properties corresponding to four queries' types: "onSpecialities", "onHealthUnits", "onDiseases" and "onComplaint" within the concept "Query". In fact, the input scenarios do not contain any information allowing our tool to recognize, automatically, the nature of the property "type". However, the expert realizes that these properties correspond to only one property "type", which corresponds to an anonymous concept with four possible values: "onSpecialities", "onHealthUnits", "onDiseases" and "onComplaint" within the Pegasus concept "Query". In contrary, the important value of recall (80,77%) and completeness (76,19%) confirm that our tool generated a good number of pertinent properties, regardless of some exceptions caused by the full automation of our approach.

Finally, the precision in terms of actions are relatively low in comparison with the concepts and the properties ones. More specifically, the precision in terms of actions for the Game of war equals 66,66%. This result is justified by the fact that our tool generates an action definition for each met action predicate within a mapping rule, especially for the human manual actions, which should not be implemented. For example, our tool generates the Pegasus actions "show(cards)", "select(card,pile)" and "select(menu-item)", although they correspond to simple mouse clicks on buttons or items done by a human actor. However, the experts do not create Pegasus actions for them because they know that they will be treated automatically by Pegasus_f, which will implement the corresponding treatment within their methods "addActionListener". We believe that this fact does not really matter while we get a good correctness value for the Game of war (83,33%) and which implies the good quality of the generated actions' implementations.

*Threats to validity.* One external threat of our approach consists of the Pegsaus project, in particular the version Pegasus_f of code generator; it has not yet been finished totally; there are still some small improvements to integrate in this project, such as treating the ellipses. However, the current version of Pegasus_f is powerful and it generates good results, shown in our evaluation. Besides, although our approach is original and treats an original topic thanks to its ability to work on any natural language, it is rather theoretic. In fact, on the practical level, we have to possess a huge number of rich dictionaries in order to parse an input instruction and deduce the corresponding mapping rule. However, the NOOJ project is always processing and many dictionaries for many languages are integrated each year. On the other hand, our tool presents an internal threat: a grammar should be created for each integrated language in order to synthesize the mapping rules. However, we believe that this is not a problem while the creation of this grammar is done only one time, then it becomes ready to treat the input instructions in that language. Another internal concern consists of the generation of an important number of concepts and actions (and thus Java classes and methods). However, we believe that this concern does not really matter because the unnecessary classes and methods generated by our approach will be latter removed by the programmer when revising the generated version of source codes. Another threat against our approach is its dependance from the input requirements. More specifically, the more complete input requirements are, the better results we get. In fact, the evaluation of our approach showed interesting results in terms of the adopted measurement values (i.e. precision, recall, F-measure...) because we have got good inputs in terms of use case scenarios. However, these values would decrease in case of a lack of information within the input requirements.

## VI. CONCLUSION AND FUTURE WORK

This paper presented a new, original approach for extracting source code from requirements written, theoretically, in any and purely natural language. Firstly, the proposed approach takes the textual descriptions (requirements) and translates them into the semantic model by extracting the corresponding mapping rules. This model gives our approach the advantage of using semantic information to explore and interpret the syntax and the semantics of the requirements. Moreover, our approach allows the translation of the mapping rules to English in case where their contents are in a language different from English. Secondly, our approach deduces a refined Pegasus code corresponding to the mapping rules based on some transformation rules. Finally, Pegasus_f translates this code into Java. Thus, the developers will save time because they are not obliged to create the initial classes of the system (including the constructors, the attributes, the getters and the setters, as well as at least the methods signatures) or to import the required packages. Our approach is implemented by the CodeRec-tool, which automates its steps.

In contrast to the existing approaches, our approach is very simple; it does not necessitate any pre-study on a particular language to be used. In fact, it accepts language-independent descriptions, understandable even by a non-IT person. In addition, another power of our approach is its ability to be used with many code generators, not necessarily Pegasus_f.

We think that the future programming techniques will follow the same direction in which a human thinks. It is our belief that the naturalistic programming will have a prominent place in the future of programming languages. The research that we presented in this paper constitutes a contribution in programming using any and purely natural language thanks to the semantic model.

In our future works, we aim to conduct an evaluation on a larger set of products to confirm the presented results. Another practical extension of the herein presented work is the application of our approach on other code generators, such as the ReDSeeDS tool, which extracts a Java code, following a Model/View/Controller architecture, from use case scenarios written in the RSL language.

## REFERENCES

[1] R. Knöll and M. Mezini, "Pegasus: First steps toward a naturalistic programming language," in Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. New York, NY, USA: ACM, 2006, pp. 542–559.

[2] L. Khaylov, "Implementation of the naturalistic programming language pegasus," Master's thesis, Darmstadt University of Technology, Germany, 2009.

[3] R. Knöll. Pegasus project. [Online]. Available: http://www.pegasus-project.org/en/Welcome.html [retrieved: August, 2017] (2006)

[4] H. Liu and H. Lieberman, "Metafor: Visualizing stories as code," in Proceedings of International Conference on Intelligent User Interfaces. New York, NY, USA: ACM, 2005, pp. 305–307.

[5] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in Proceedings of ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. New York, NY, USA: ACM, 2015, pp. 416–432.

[6] J. Franců and P. Hnětynka, Automated generation of implementation from textual system requirements. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 34–47.

[7] M. Smialek and W. Nowakowski, Introducing requirements-driven modelling. Switzerland: Springer International Publishing, 2015, ch. From Requirements to Java in a Snap, pp. 1–30.

[8] R. Knöll, V. Gasiunas, and M. Mezini, "Naturalistic types," in Proceedings of SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. New York, NY, USA: ACM, 2011, pp. 33–48.

[9] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Feature model extraction from documented uml use case diagrams," Ada User Journal, vol. 35, no. 2, 2014, pp. 108–117.

[10] ——, "Mining feature models from functional requirements," Computer journal, vol. 59, no. 7, 2016, pp. 1–21.

[11] M. Silberztein, Formalizing natural languages: The NooJ approach. John Wiley Sons, Inc., 2016.

[12] G. Glavas and J. Snajder, "Construction and evaluation of event graphs," Natural Language Engineering, vol. 21, no. 4, 2015, pp. 607–652.

[13] C. J. Fillmore, "Frame semantics and the nature of language," in Origins and evolution of language and speech, S. Harnad, Ed. Academy of Sciences, 1976, pp. 155–202.

[14] C. J. Fillmore and B. T. Atkins, Towards a frame-based lexicon: The semantics of RISK and its neighbors. Hillsdale: Lawrence Erlbaum Associates, 1992, pp. 75–102.

[15] C. J. Fillmore and B. Collin, A frames approach to semantic analysis. Oxford: Oxford University Press, 2010, pp. 313–339.

[16] J. Ruppenhofer, M. Ellsworth, M. R. L. Petruck, C. R. Johnson, and J. Scheffczyk. Framenet II: Extended theory and practice. [Online]. Available: http://framenet.icsi.berkeley.edu [retrieved: July, 2017] (2010)

[17] C. F. Baker, C. J. Fillmore, and J. B. Lowe, "The Berkeley FrameNet project," in Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1, ser. ACL '98. Stroudsburg, PA, USA: Association for Computational Linguistics, 1998, pp. 86–90. [Online]. Available: http://dx.doi.org/10.3115/980845.980860

[18] Marie-Claude L'Homme, "Terminologie de l'environnement et sémantique des cadres," Congrès Mondial de Linguistique Franaise, SHS Web of Conferences, vol. 27, 2016, pp. 1–14.

[19] T. Schmidt, "The kicktionary a multilingual lexical resource of football language," in Multilingual FrameNets in computational lexicography : methods and applications, H. C. Boas, Ed., 2009.

[20] A. Dolbey, M. Ellsworth, and J. Scheffczyk, "Bioframenet: A domain-specific framenet extension with links to biomedical ontologies," in In Proceedings of the Biomedical Ontology in Action Workshop at KR-MED, 2006, pp. 87–94.

[21] J. Pimentel, "Description de verbes juridiques au moyen de la sémantique des cadres," in Terminologie and Ontologie : Théories et applications, 2010, pp. 26–27.

[22] M. Smialek, W. Nowakowski, N. Jarzebowski, and A. Ambroziewicz, "From use cases and their relationships to code," in International Workshop on Model-Driven Requirements Engineering, Chicago, IL, USA, September 24, 2012, pp. 9–18.

[23] W. Nowakowski, M. Smialek, A. Ambroziewicz, and T. Straszak, "Requirements-level language and tools for capturing software system essence," Comput. Sci. Inf. Syst., vol. 10, no. 4, 2013, pp. 1499–1524.

[24] A. Kalnins, et al., Handbook of research on innovations in systems and software engineering. IGI Global, 2014, ch. Developing Software with Domain-Driven Model Reuse.

[25] A. Cozzie and S. T. King, "Macho: Writing programs with natural language and examples," University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[26] E. Özcan, S. E. Seker, and Z. I. Karadeniz, "Generating java class skeleton using a natural language interface," in Natural Language Understanding and Cognitive Science, Porto, Portugal, April 2004, 2004, pp. 126–134.

[27] M. Kapor. Brainy quote. [Online]. Available: http://www.brainyquote.com/quotes/quotes/m/mitchkapor690403.html [retrieved: August, 2017] (1950)

[28] R. S. Scowen, "Extended BNF - A generic base standard," in Proceedings of the 1993 Software Engineering Standards Symposium (SESS'93), Aug. 1993.

[29] M. Mefteh. Requirements analysis with the semantic model. [On-

line]. Available: http://spl-nlp-with-semanticmodel.com/translation.html [retrieved: August, 2017] (2017)

[30] J. Ramos, "Using TF-IDF to determine word relevance in document queries," Department of Computer Science, Rutgers University, 23515 BPO Way, Piscataway, NJ, 08855e, Tech. Rep., 2003.

[31] G. A. Miller. Wordnet. [Online]. Available: https://wordnet.princeton.edu/ [retrieved: August, 2017] (2015)

[32] M. Ester, H. peter Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in Proceedings of the International Conference on Knowledge Discovery and Data Mining. AAAI Press, 1996, pp. 226–231.

[33] J. Trimble. et al., Google translate API. [Online]. Available: https://www.programmableweb.com/api/google-translate [retrieved: August, 2017] (2011)

[34] C. Wohlin, M. Höst, and K. Henningsson, Empirical research methods in web and software engineering. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 409–430.

[35] L. P. Tizzei, C. M. F. Rubira, J. Lee, A. Garcia, and M. Barros. Health complaint system. [Online]. Available: http://www.ic.unicamp.br/ tizzei/phc/jss2013/ [retrieved: August, 2017] (2013)

[36] K. Bittner and I. Spence, Use case modeling. Pearson Education Inc., 2002, pp. 301–330.

[37] D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid, "Gophone - a software product line in the mobile phone domain," No. 025.04/E, Version 1.0, Fraunhofer IESE, Tech. Rep., 2004.

[38] J. Kienzle, N. Guelfi, and S. Mustafiz, Crisis management systems: A case study for aspect-oriented modeling. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–22.

[39] G. Blank. Game of war. [Online]. Available: http://www.cse.lehigh.edu/ glennb/csc10/WarDesign.htm [retrieved: June, 2017] (2010)

[40] C. R. van der Burg, T. Kirke, and A. Rokic, "Emptio - a mobile phone application for selfservice," Master's thesis, Aalborg University, Germany, 2011.