Detecting Feature Duplication in Natural Language Specifications when Evolving Software Product Lines

Amal Khtira, Anissa Benlarabi and Bouchra El Asri
IMS Team, SIME Laboratory, ENSIAS, Mohammed V University, Rabat, Morocco

Keywords: Natural Language Requirements, Software Product Line, Feature Duplication, Natural Language Processing.

Abstract: Software product lines are dynamic systems that need to evolve continuously to meet new customer requirements. This evolution impacts both the core platform of the product line and its derived products. For several reasons, the most common way to express requirements by customers is natural language. However, the experience has shown that this communication channel does not give the possibility to detect system defects such as inconsistency and duplication. The objective of this paper is to propose a method to transform textual requirements into the XML format used by some Feature-oriented software development tools, in order to facilitate the detection of features duplication.

1 INTRODUCTION

Software Product Line Engineering (SPLE) (Clements and Northrop, 2002) is an approach that aims at reusing common assets to generate customized applications according to different needs of customers. The main benefits of this approach are the cost reduction, quality enhancement and time to market reduction. The SPLE approach involves two major processes, domain engineering and application engineering (Pohl et al., 2005). Domain engineering consists in defining the variability and commonality of the product line, and building all reusable assets (i. e. requirements, architecture, components, tests). Based on these assets, specific applications are derived during the application engineering process.

Software product lines are a long-term investment and have to evolve constantly in response to business needs, changing markets, and advances in technology. This evolution impacts both the common assets of the product line and the specific assets of individual applications. Due to this change, several defects can arise in the SPL models, such as the inconsistency and incompleteness of features (Reder and Egyed, 2013)(Zowghi and Gervasi, 2003), and the non-conformance of constraints (Mazo et al., 2011).

What makes the evolution process more complex is that, in most current software projects, requirements documents are written in natural language because it is the simplest and the more flexible way for customers to express their expectations. However, specifying requirements in natural language has shown many drawbacks such as imprecision, inaccuracy and ambiguity (Meyer, 1985)(Lami et al., 2004). In order to preserve the consistency and correctness of the SPL models, many approaches have been proposed to manage the evolution process by verifying the specifications of change requests considered as the main source of model defects. A large number of these approaches aimed at transforming the natural language requirements to a formal or semi-formal representation (Lami et al., 2004)(Kamalrudin et al., 2010)(Holtmann et al., 2011).

In this paper, we propose a method based on natural language processing to transform natural language requirements into a tree-like document (i. e. XML document). This document will serve as an input to an algorithm whose purpose is to detect a specific defect in the specification, the feature duplication (Khtira et al., 2014). We consider that two features are duplicated when they satisfy the same functionality in the product line.

The remainder of the paper is organized as follows. Section 2 gives an insight of the knowledge base of our work. In Section 3, we explain our method of transforming natural language specifications into XML documents, and we present the algorithm of detection of feature duplication. Section 4 provides an overview of different works of the literature related to our approach and explains in what our contribution is different. In section 5, we conclude and outline future work.

Khtira A., Benlarabi A. and El Asri B..
Detecting Feature Duplication in Natural Language Specifications when Evolving Software Product Lines.
DOI: 10.5220/0005459802570262
In *Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2015), pages 257-262
ISBN: 978-989-758-100-7

2 BACKGROUND

In this section, we introduce the background of our study. First, we give an overview of the feature duplication problem. Then, we discuss the transformation of requirements from an informal to a formal representation.

2.1 Feature Duplication

Software product lines are dynamic systems that need to evolve constantly to address changes such as new customer requirements, technology changes, or refactoring. The possible changes in a product line involve changes that affect the entire product line or changes driven by an individual product. These changes can be the source of many defects in the product line such as inconsistency (Reder and Egyed, 2013)(Blanc et al., 2009), incompleteness (Zowghi and Gervasi, 2003) and duplication (Khtira et al., 2014). Duplication is the fact of adding features that have the same role in the application, which means that they satisfy the same functionality.

In (Thomas and Hunt, 1999), Hunt and Thomas distinguish the four main causes of duplication, namely imposed duplication, inadvertent duplication, impatient duplication and inter-developer duplication. In the case of software product lines, at least the two last causes might occur. Indeed, due to tight deadlines, developers could possibly get impatient and add duplicated features from the specifications. In addition, the large number of customers and developers working on the system can easily cause feature duplication. Many studies have dealt with defects in SPL such as inconsistency and incompleteness, but to the best of our knowledge, few attempts have tackled the problem of duplication. In our study, we will focus on this specific defect. Since manual verification has proved to be time-consuming and error prone, we propose an automatic algorithm to detect duplication based on the formal presentation generated from a textual requirements of an evolution.

2.2 Formalizing Natural Language Requirements

In most development projects, textual requirements are an important input of the requirements analysis. Although formal methods are used in some safety-critical systems, for the largest part of software projects, the predominant mean of representing requirements is Natural Language due to many reasons. Indeed, specifying requirements in natural language is more flexible and simple for the customer. Moreover, natural language can be understood by all the stakeholders of the project, unlike formal methods which can be very difficult to customers. Hence, natural language specifications can be used as an agreement between customers and suppliers and as an output for the project management. However, expressing requirements in a natural language frequently make them prone to many defects. For instance, (Meyer, 1985) details seven problems with natural language specifications: noise, silence, over-specification, contradictions, ambiguity, forward references and wishful thinking. (Lami et al., 2004) dealt with other defects, namely the ambiguity, the inconsistency and the incompleteness. To overcome these problems, many studies have proposed methods to transform natural language specifications to formal or semi-formal specifications (Holtmann et al., 2011)(Fatwanto, 2013)(Cabral and Sampaio, 2008)(Ilieva and Ormandjieva, 2005).

In the context of software product lines, the domain model of the entire product line and the application models of the derived products can be expressed using feature models, while the specifications of a new evolution concerning a derived product can be expressed using natural language. To enable a verification of the specifications against the product line models, we need first to transform these requirements into a more formal presentation and to assure the absence of all sorts of defects.

3 DUPLICATION DETECTION IN NATURAL LANGUAGE SPECIFICATIONS

The purpose of our work is to detect duplication in textual specifications related to an evolution of a software product line. To achieve this goal, we propose a two-step process. The first step consists of transforming the natural language specifications into an XML document. The second step involves the application of an algorithm that detects duplicated features in the generated XML. Figure 1 represents the overview of the proposed process.

3.1 Initiation Phase

The objective of this phase is to initiate the model that will be used by the machine learning. The model is thus populated based on the domain model, which contains all the existing features implemented by the product line. As a case study of our approach, we consider a CRM (Customer Relationship Management)

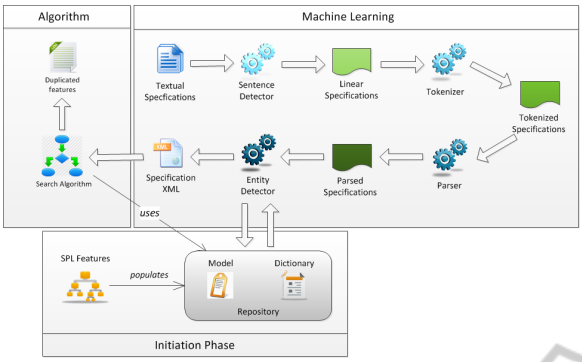


Figure 1: An overview of the process.

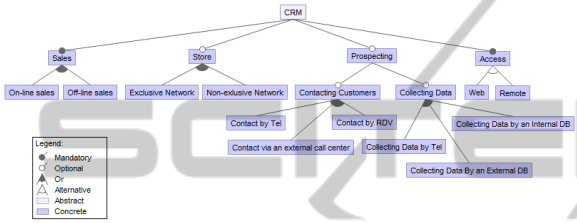


Figure 2: The Domain Feature Model of the CRM.

product line with the feature model depicted in Figure 2.

It has to be noted that the domain model and the application models of the SPL are created using the FeatureIDE tool (Kastner et al., 2009), which is an open source framework for software product line engineering based on Feature-Oriented Software Development (FOSD). This framework supports the entire life-cycle of a product line, especially domain analysis and feature modeling. It provides a graphical presentation of the feature model tree and generates automatically the corresponding XML source. This structure of the XML distinguishes the concepts of variation points and variants using tags. The tags ”or” and ”alt” correspond to variation points, while the tags ”feature” correspond to variants.

Expressing the feature models using an XML format will allow us to anticipate the comparison between these models and the specifications of evolutions in terms of duplication.

3.2 Formalizing Natural Language Specifications

In this section, we explain the process of transformation of natural language requirements into an XML document. To perform this operation, we will use the OpenNLP library (OpenNLP, 2011), which is a machine learning based toolkit for the processing of natural language text. The remainder of this subsection details the different steps and artifacts involved in this process.

3.2.1 Textual Specification

In this stage of the process, the main input is the specification of a new evolution related to a derived product. This specification contains the new requirements that have to be implemented in this specific product. The easiest and more flexible way for the customer to express his requirements is natural language. Hence, the specification can be written in a document with a doc or txt format.

3.2.2 Sentence Detector

The first step of this stage consists of detecting the punctuation characters that indicate the end of sentences. After the detection of all sentence boundaries, each sentence is written in its own line. By processing the input textual specification, the output of this operation will be a text document that contains a sentence per line.

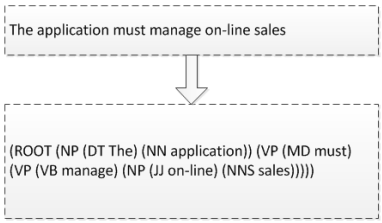
3.2.3 Tokenizer

This step consists of segmenting the resulted sentences of the previous step into tokens. A token can be a word, a punctuation, a number, etc. As an output of this action, all the tokens of the specification are separated using whitespace characters, such as a space or line break, or by punctuation characters.

3.2.4 Parser

The parser is responsible for analyzing each sentence of the specification in order to mark tokens with their corresponding types and roles in the sentence based on the rules of the language grammar (e.g. noun, verb, adjective, adverb). We note that the language used in our input specifications is English. A parser marks all the words of a sentence using a POS tagger (Part-Of-Speach tagger) and converts the sentence into a tree that represents the sentences syntactic structure. Figure 3 depicts an example of parsing for a textual definition of a requirement related to the case study presented in Figure 2.

This operation enables us to have an exact understanding of the sentence. For example, it allows us to confirm whether the action of a verb is negative or affirmative, and whether a requirement is mandatory or optional.



DT	Definite Article
NN	Noun
MD	Model Verb
VB	Verb
JJ	Adjective
NNS	Noun (Plural)

Figure 3: An example of syntax parsing.

3.2.5 Entity Detector

The aim of this step is to detect semantic entities in the specification. In our study, we are interested especially in the parts of the sentences considered as variation points and variants. To carry out this task, we need the model created in the initiation phase where all the domain specifications are tagged.

In the example depicted in Figure 4, the entity detector reads a tokenized sentence and outputs the sentence with markup for the detected variation point and variant.

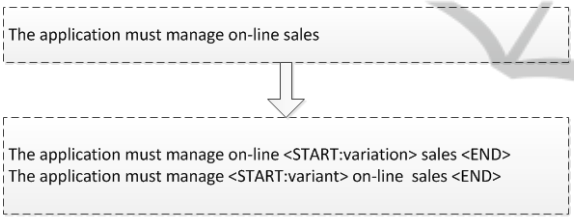


Figure 4: An example of detecting entities.

In order to measure the precision of entity recognition, we use the evaluation tool of OpenNLP that provides information about the accuracy of the used model.

3.2.6 Repository

The repository contains two main components:

- The model that contains the different features of the domain model, classified in different categories, especially <variation point> and <variant>.
- The dictionary which contains the set of synonyms and alternatives for all the concepts used in the system.

The repository is initially populated based on the domain model of the product line. So that the repository keeps up with the evolution of the product line and its derived products, the new concepts detected in the specification are added to the initial repository, which makes the latter more accurate.

3.2.7 XML Output

The output of our process is the specification in the form of an XML document. The XML should contain the tags ”or” and ”alt” for variation points, and the tags ”feature” for variants. This structure provides a unified presentation for both the domain model and the specification. The XML document will serve as an input of the algorithm for duplication detection.

3.3 The Algorithm to Detect Feature Duplication

With the large number of features in a specification, detecting duplication manually becomes a tedious, time-consuming and error-prone task. In order to identify efficiently the duplicated features, we propose an automatic algorithm that uses as an input the generated XML of the specification. An example of this XML is depicted in Figure 5.

```
<struct>
    <and abstract="true" mandatory="true" name="CRM1">
        <or mandatory="true" name="Sales">
            <feature mandatory="true" name="On-line sales"/>
            <feature mandatory="true" name="Off-line sales"/>
        </or>
        <or name="Store">
            <feature mandatory="true" name="Exclusive network"/>
            <feature mandatory="true" name="Non-exclusive network"/>
        </or>
        <and name="Prospecting">
            <or mandatory="true" name="Collecting Data">
                <feature mandatory="true" name="Collecting Data by Tel"/>
                <feature mandatory="true" name="Collecting Data by an external DB"/>
                <feature mandatory="true" name="Collecting Data by an internal DB"/>
            </or>
        </and>
        <and mandatory="true" name="Access">
            <feature name="Web"/>
        </and>
    </and>
</struct>
```

Figure 5: An example of the algorithm input.

We denote by S the XML of the specification. PA is the set of nodes related to the tags ”or” and ”alt” of S (i. e. variation points), and V is the set of nodes related to the tags ”feature” of S (i. e. variants).

$$P = \{p_1, p_2, \ldots, p_n\}$$

$$\forall p_i \in P \quad \exists V_i \text{ where } V_i = \{v_{ij} \quad | \quad j \in \mathbb{N}\}$$

Thus:

$$V = \bigcup_{i=1}^{n} V_i$$

The proposed algorithm contains the following steps:

- **Step 1.** This step is based on the dictionary that contains the synonyms of all the concepts of the SPL, and for each set of synonyms, we define a key synonym. For example : The synonyms for ”on-line sales” could be ”e-sales”, ”Internet sales”, or ”web sales”. The key synonym for these alternatives is ”on-line sales”.

  The aim of this step is to generate an equivalent XML of the input, by replacing the name of every

node (variation point or variant) with its associated key synonym in the dictionary.

- **Step 2.** This step consists of putting in alphabetical order the variation points and the variants of each variation point.
- **Step 3.** For each variation point, the duplicated variants are detected and removed from the XML. The sub-algorithm of this step is as follows:

Algorithm 1: Detecting duplicated variants of a variation point.

```
// n_k the number of variants of the variation point p_k
// DP the set of variation points with duplicated variants
// DV_j the set of duplicated variants of the variation point
p_j
DP = ∅
for each p_k ∈ P do
  DV_k = ∅
  i ← 0
  while i < n_k do
    if v_ki = v_ki+1 then
      DP ← DP ∪ {p_k}
      DV_k ← DV_k ∪ {v_ki}
      V_k ← V_k \ {v_ki}
    end if
    i ← i + 1
  end while
end for
```

- **Step 4.** During this step, we carry out a comparison between the variants of all the variation points, in order to detect duplication in the whole XML.

The final output of these steps is a log file that contains the set of duplicated pairs (variation point, variant) of the specification. These features are sent to the user in order to reverify his needs and change them in case of error.

4 RELATED WORK

Several papers have dealt with detecting defects in natural language specifications. In this section, we provide an insight of the studies that use the transformation of natural language requirements into a formal or semi-formal presentation.

Lami et al. (Lami et al., 2004) propose a methodology and a tool called QuARS (Quality Analyzer for Requirement Specifications). This tool performs an initial parsing of the specifications in order to detect automatically specific linguistic defects, namely inconsistency, incompleteness and ambiguity. The analysis performed by QuARS is limited to syntax-related issues of a natural language document, while semantic-related problems are not directly addressed. Although this study converges with our paper in the fact that it is based on parsing of natural language requirements to detect defects. However, our approach is different because it transforms specifications into XML trees and it focuses particularly on a specific defect which is the feature duplication.

Kamalrudin et al. (Kamalrudin et al., 2010) presented the automated tracing tool Marama that enables users to capture their requirements and automatically generate the Essential Use Cases (EUC). This tool supports the inconsistency checking between the textual requirements, the abstract interactions and the EUCs, but unlike our approach, this one is based on use cases instead of XML documents.

In order to locate inconsistency in the domain feature model of a SPL, Yu (Yu et al., 2012) provides a new method to construct traceability between requirements and features. It consists of creating individual application Feature Tree Models (AFTMs) and establishing traceability between each AFTM and its corresponding requirements. It finally merges all the AFTMs to extract the Domain Feature Tree Model (DFTM), which enables to figure out the traceability between domain requirements and DFTM. Using this method helps constructing automatically the domain feature model from requirements. It also helps locate affected requirements while features change or vice versa, which makes it easier to detect inconsistencies. However, this approach is different from our own one, because we suppose that domain and application models exist, our objective is hence to construct a more formal presentation of the requirements related to a new version of a derived product.

At the aim of validating and correcting automatically textual requirements, Holtmann et al. (Holtmann et al., 2011) proposed an approach that uses an extended CNL (controlled natural language) that is already used in the automotive industry. The CNL requirements are first translated into an ASG (Abstract Syntax Graph) typed by a requirements metamodel. Then, structural patterns are specified based on this metamodel. The use of patterns allows an automated correction of some requirements errors and the validation of requirements due to change requests. While this approach considers the correction of textual specifications using a CNL, our approach aims at detecting duplication in these specifications by transforming them into tree-like documents.

Similarly, Cabral and Sampaio (Cabral and Sampaio, 2008) proposed a novel approach that uses templates to write use cases in CNL (Controlled Natural Language), then to translate use cases in CNL to models in CSP process algebra that allows the de-

scription of systems in terms of processes that operate independently, and interact with each other through message-passing communication. The use of a CNL and use case templates enables to guarantee requirements consistency. The paper focuses on use case transformation but does not detail the process of inconsistency detection.

5 CONCLUSION

In this paper, we presented an approach to detect duplication in natural language specifications related to a derived product of a software product line. This approach is based on a two-phase process. During the first phase, the specifications are transformed into a more formal presentation, XML document, using natural language processing. In the second phase, we propose an algorithm that searches the duplicated features in the generated XML. Apart from the detection of duplication within the specification, the formalization of the latter anticipates the verification of duplication between the new requirements and the existing domain and application models of the software product line.

As a case study for our work, we considered a CRM SPL. Work in progress consists of creating a model based on the domain specifications of the CRM tool and using the OpenNLP library to validate it. We also prepare the specification of a new evolution which will serve as an input of the proposed process. In a future work, we intend to develop a support tool whose objective is to apply the syntax parsing to the specification, generate the XML and apply the algorithm to detect automatically the duplicated features.

REFERENCES

Blanc, X., Mougenot, A., Mounier, I., and Mens, T. (2009). Incremental detection of model inconsistencies based on model operations. In *Advanced information systems engineering*, pages 32–46. Springer.

Cabral, G. and Sampaio, A. (2008). Formal specification generation from requirement documents. *Electronic Notes in Theoretical Computer Science*, 195:171–188.

Clements, P. and Northrop, L. (2002). *Software product lines: practices and patterns*, volume 59. Addison-Wesley Reading.

Fatwanto, A. (2013). Software requirements specification analysis using natural language processing technique. In *QiR (Quality in Research), 2013 International Conference on*, pages 105–110. IEEE.

Holtmann, J., Meyer, J., and von Detten, M. (2011). Automatic validation and correction of formalized, textual requirements. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 486–495. IEEE.

Ilieva, M. and Ormandjieva, O. (2005). Automatic transition of natural language software requirements specification into formal presentation. In *Natural Language Processing and Information Systems*, pages 392–397. Springer.

Kamalrudin, M., Grundy, J., and Hosking, J. (2010). Managing consistency between textual requirements, abstract interactions and essential use cases. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual*, pages 327–336. IEEE.

Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., and Apel, S. (2009). Featureide: A tool framework for feature-oriented software development. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 611–614. IEEE.

Khtira, A., Benlarabi, A., and El Asri, B. (2014). Towards duplication-free feature models when evolving software product lines. In *Software Engineering Advances (ICSEA), 2014 Ninth International Conference on*, pages 107–113.

Lami, G., Gnesi, S., Fabbrini, F., Fusani, M., and Trentanni, G. (2004). An automatic tool for the analysis of natural language requirements. *Informe técnico, CNR Information Science and Technology Institute, Pisa, Italia, Setiembre*.

Mazo, R., Lopez-Herrejon, R. E., Salinesi, C., Diaz, D., and Egyed, A. (2011). Conformance checking with constraint logic programming: The case of feature models. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 456–465. IEEE.

Meyer, B. (1985). On formalism in specifications. *IEEE software*, 2(1):6–26.

OpenNLP (2011). Apache software foundation. *URL http://opennlp. apache. org*.

Pohl, K., Böckle, G., and Van Der Linden, F. (2005). Software product line engineering. *Springer*, 10:3–540.

Reder, A. and Egyed, A. (2013). Determining the cause of a design model inconsistency. *Software Engineering, IEEE Transactions on*, 39(11):1531–1548.

Thomas, D. and Hunt, A. (1999). The pragmatic programmer: From journeyman to master.

Yu, D., Geng, P., and Wu, W. (2012). Constructing traceability between features and requirements for software product line engineering. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 2, pages 27–34. IEEE.

Zowghi, D. and Gervasi, V. (2003). On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology*, 45(14):993–1009.