Information **2015**, *6*, 592-612; doi:10.3390/info6040592

OPEN ACCESS

information

ISSN 2078-2489

www.mdpi.com/journal/information

Article

Duplication Detection When Evolving Feature Models of Software Product Lines

Amal Khtira *, Anissa Benlarabi † and Bouchra El Asri †

IMS Team, SIME Laboratory, ENSIAS, Mohammed V University, Rabat 10000, Morocco;
E-Mails: a.benlarabi@gmail.com (A.B.); elasri_b@yahoo.fr (B.E.A.)

† These authors contributed equally to this work.

* Author to whom correspondence should be addressed; E-Mail: amalkhtira@gmail.com;
Tel.: +212-67-1107452.

Academic Editor: Ahmed El Oualkadi

Received: 23 June 2015 / Accepted: 24 September 2015 / Published: 7 October 2015

Abstract: After the derivation of specific applications from a software product line, the applications keep evolving with respect to new customer's requirements. In general, evolutions in most industrial projects are expressed using natural language, because it is the easiest and the most flexible way for customers to express their needs. However, the use of this means of communication has shown its limits in detecting defects, such as inconsistency and duplication, when evolving the existing models of the software product line. The aim of this paper is to transform the natural language specifications of new evolutions into a more formal representation using natural language processing. Then, an algorithm is proposed to automatically detect duplication between these specifications and the existing product line feature models. In order to instantiate the proposed solution, a tool is developed to automatize the two operations.

Keywords: software product line; natural language specifications; feature models; evolution; duplication; natural language processing

1. Introduction

Due to the insistent demand for software customization and the aim of reducing the cost of development, reducing the time-to-market and improving the quality of products, a switch from developing individual software to developing a family of software began, resulting in software product line engineering (SPLE) [1]. This approach consists of deriving specific software from a core platform.

Many SPL-related paradigms have been proposed, among which feature-oriented software development (FOSD) [2] is an approach that structures the design and code of a software system using the concept of a feature. The goal of FOSD is to decompose a software system in terms of the features it provides and to create many different software products that share common features and differ in other features. The commonalities and variabilities of a product line can be expressed using feature-oriented domain analysis (FODA) feature model. FODA [3] is a domain analysis method that provides a complete description of the features of the domain, giving less attention to design and code.

A product line is a long-living system that must continuously evolve to satisfy the new needs of customers. The evolution of an SPL impacts both the domain model of the product line and the application models of derived products. These changes can be the source of several defects addressed in many papers in the literature [4,5]. Since the requirements related to SPL evolutions are most of the time expressed in the form of natural language specifications, the verification of the model becomes more complex. To solve this problem, it is necessary to transform the natural language specifications into a more formal representation in order to simplify the detection of defects. In this paper, we focus on a specific model defect, *i.e.*, feature duplication [6]. We consider duplication the fact of adding to the domain or the application model features with the same semantics, which means that they satisfy the same functionality.

Thus, the aim of this paper is two-fold. First, we transform natural language specifications into a formal representation using a natural language processing approach. Second, we propose algorithms to detect duplication when introducing features from the specifications of new evolutions into the existing SPL feature models. To automatize the two operations, we provide an automated tool that enables avoiding the complexity of manual verification.

The remainder of the paper is organized as follows. Section 2 gives an overview of the background of our study by presenting some issues related to software product lines' evolution. In Section 3, we describe the methods to model and document software product lines and its evolutions, and we explain our approach to verify the evolved SPL models. In Section 4, we propose a framework to detect feature duplication when evolving software product lines and we present the automated tool developed to automatize the framework processes. In Section 5, we illustrate our approach through a case study, and we discuss the results achieved to date. Section 6 positions our approach with related works. The paper is concluded in Section 7.

2. Software Product Line Evolution

In this section, we introduce the background of our study. First, we present the SPLE approach and the evolution challenges related to SPLs. Then, we discuss model defects caused by SPL evolution, and we justify our focus on feature duplication.

2.1. Software Product Lines

Originally, software have been developed individually to solve a specific problem or to satisfy the needs of a particular customer. However, in the past few years, companies started shifting to a new approach called software product line engineering.

In [1], Clements and Northop define an SPL as "*a set of intensive-software systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*". The SPLE approach [7] promotes systematic reuse throughout the software development process and allows the generation of a set of software products with a considerable gain in terms of time, cost and quality.

The domain engineering phase of the SPLE framework is responsible for capturing the common features of all of the applications, which increases the reusability of the system, and for determining the variant features, which enables the production of a multitude of specific applications that respond to different customers' requirements. During this phase, all of the shared assets of the product line are produced, starting by defining the model and then by generating the design and code of the product line.

The goal of the second process, the application engineering, is to build customized products based on the SPL. This operation is referred to as product derivation or product instantiation. Product derivation consists of selecting the features that are meaningful to a specific customer in order to create a specific software product. This process is also responsible for instantiating all of the artifacts of the product line, *i.e.*, model, design, components, *etc.*

2.2. SPL Evolution Challenges

Software product lines are long-living systems and, thus, require inevitably continuous changes to product line models. Changes can be introduced by new technologies, new customer requirements, or new business strategies. Since the quality of a product line depends on the importance attached to its evolution process, many studies in the literature have dealt with different challenges related to the evolution of software product lines.

Some examples of evolution challenges are: the use of traceability links and dependencies between the software product line artifacts in order to analyze the change impacts [8], the proposition of evolution models to manage the evolution process rigorously [9], the co-evolution of domain and application models [10], the co-evolution of different artifacts of an evolving software product line [11] and the assessment of evolution before its implementation to anticipate the defects and unexpected behaviors [12]. In our study, we focus on a specific challenge, which is the verification of SPL models during evolution and the detection and correction of the resulting model defects.

2.3. Model Defects Caused by SPL Evolution

As a consequence of software product line evolution, many problems occur in the different assets of the product line, especially model defects. This issue was the main input of our state of the art. Indeed, in the beginning of our work, we carried out a literature review whose objective was to list the different model defects caused by SPL evolution, to identify the approaches that propose solutions to these defects and to determine the limitations of these approaches. As a result of the review, we list the model defects, as presented in Table 1.

Table 1. The SPL model defects.

Model Defect	**Definition**
Inconsistency	A contradiction between two or more features, requirements or functionalities [13]. The breaking of a rule between two partial specifications [14].
Incompleteness	The lack of necessary information related to a feature or requirement [15].
Incorrectness	This describes the non-correspondence of a feature with the requirements imposed in the specification [5].
Ambiguity	When a feature has more than one interpretation [15].
Redundancy	The presence of reusable elements and variability constraints among them that can be omitted from the product line model (PLM) without loss of semantics on the PLM [16].
Duplication	To have the same thing expressed in two or more places; duplication can happen in specifications, processes and programs [17].
Unsafety	This happens when the behavior of existing products is affected by a new evolution [18].

2.4. Duplication of Features When Evolving SPLs

A review of the literature has shown that, in comparison to other defects, feature duplication is so far the one that has received little attention. According to [17], this defect occurs for many reasons, such as the non-synchronization between the different people working on the project and the rapid implementation of requirements without a verification of the existing models. Moreover, duplication addressed in the literature most of the time concerns the code level or what is called code cloning, which is a late stage of the implementation process.

Several motivations have led us to focus on the problem of feature duplication. First, the introduction of duplication prevents one from meeting the main objectives of a software product line (e.g., the reduction of time-to-market, the reduction of cost, the improvement of product quality). On the contrary, this defect causes a waste of time, money and effort by implementing the same functionalities many times. In addition, duplicate features can evolve independently from each other, which may cause inconsistencies in the model. For instance, a feature can be deleted or modified while its copy in another place of the model remains the same, which leads to a contradiction. Moreover, duplication in the feature level negatively impacts the quality of the final product by causing code clones and consequently results in the recurring bug problem and increases the maintenance effort [19]. A solution is thus necessary to detect duplicate features in an early stage of evolution, which helps avoid their inclusion in the existing models from the very beginning.

3. Modeling and Verifying SPL Features

In this section, we describe the methods to model and document software product lines and their evolutions. Then, we explain our approach to verify the evolved SPL models.

3.1. Variability Modeling

During the domain engineering of a software product line, the common and variant features of all of the specific applications are captured. To document and model variability, many approaches have been proposed. For instance, [7] introduced the orthogonal variability model, which defines variability in a separate way. Salinesi *et al.* [20] used a constraint-based product line language. Other approaches are proposed to model variability using UML models (Unified Modeling Language models) or feature models (FODA [3]). In our study, we opt for the FODA method. The FODA method was introduced in 1990 as a means of representing the commonalities and variabilities of software product lines using feature models (FMs). A feature, as described by [21,22], is the abstraction of functional or non-functional requirements that help characterize the system and that must be implemented, tested, delivered and maintained. A feature model is the description of all of the possible features of a software product line and the relationships between them. It is depicted as a tree-like structure where a feature is represented by a node and sub-features by children nodes [1].

Feature-oriented software development (FOSD) [2] is a development paradigm that has his roots in the FODA method. This paradigm aims at automatically generating software products based on the feature models. Hence, tools, such as FeatureIDE [23], have been proposed to formalize the representation of feature models and to enable the automatic selection of features of derived products. This tool will be used in our approach to formalize the SPL feature models.

3.2. Evolution Documentation

Similarly to other software systems, the evolutions of a product line and its derived products are most of the time expressed in the form of natural language specifications. Indeed, natural language is the preferable way for customers to express what they expect from the system and to explain their perception of the problem easily. However, expressing requirements in a natural language frequently makes them prone to many defects. In [24], Meyer details seven problems with natural language specifications: noise, silence, over-specification, contradictions, ambiguity, forward references and wishful thinking. Lami [15] dealt with other defects, namely the ambiguity, the inconsistency and the incompleteness. In order to be able to detect defects introduced by specifications, many studies have proposed methods to transform natural language specifications to formal or semi-formal specifications [25–27].

3.3. SPL Model Verification

On the one hand, the domain model of a product line is the representation of all of the different and common features of a family of related products. On the other hand, the application model is the model corresponding to an individual application and is generated by binding the variability of the domain model in a way that satisfies the needs of a specific customer [7]. These two models 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 to unify the presentation of these two inputs. Hence, the objective of our study is to transform the natural language requirements of an evolution and the feature models of the product line into a unified representation. The final representation has to be formal to allow the automatic model verification and the detection of model defects, especially duplication.

4. A Framework for Detecting Feature Duplication When Evolving SPLs

The aim of our approach is to detect duplication when implementing an evolution of a derived product. To achieve this goal, we propose a framework with two main processes [28]. The first process consists of unifying the form of the evolution specifications and the feature models related to the SPL. The second process involves the detection of duplicated features in the new evolution. Figure 1 represents the overview of the proposed framework. In the following sub-sections, we explain each process in detail. Then, we propose a support tool whose objective is to automatize the framework processes.

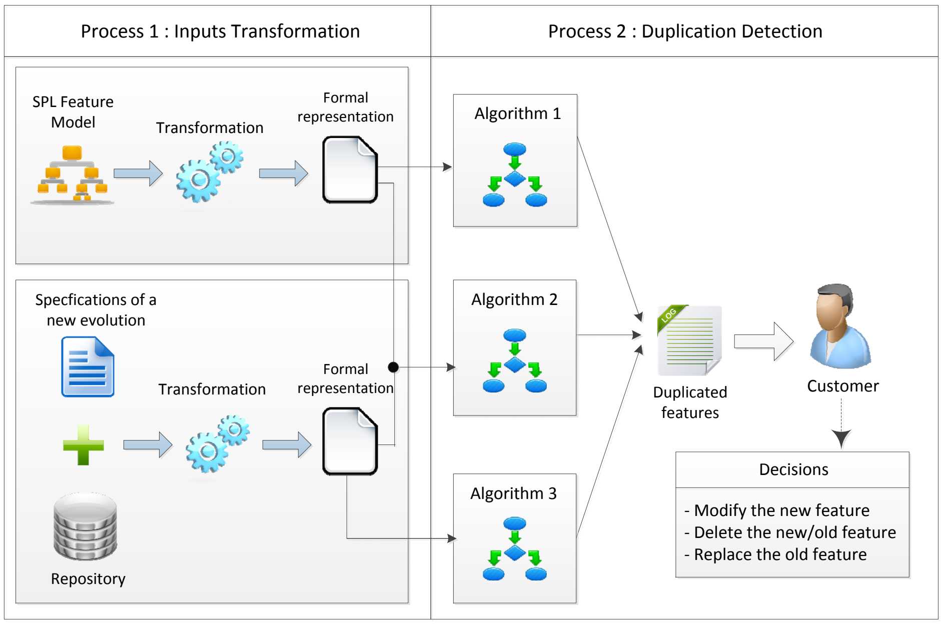


Figure 1. The overview of the framework.

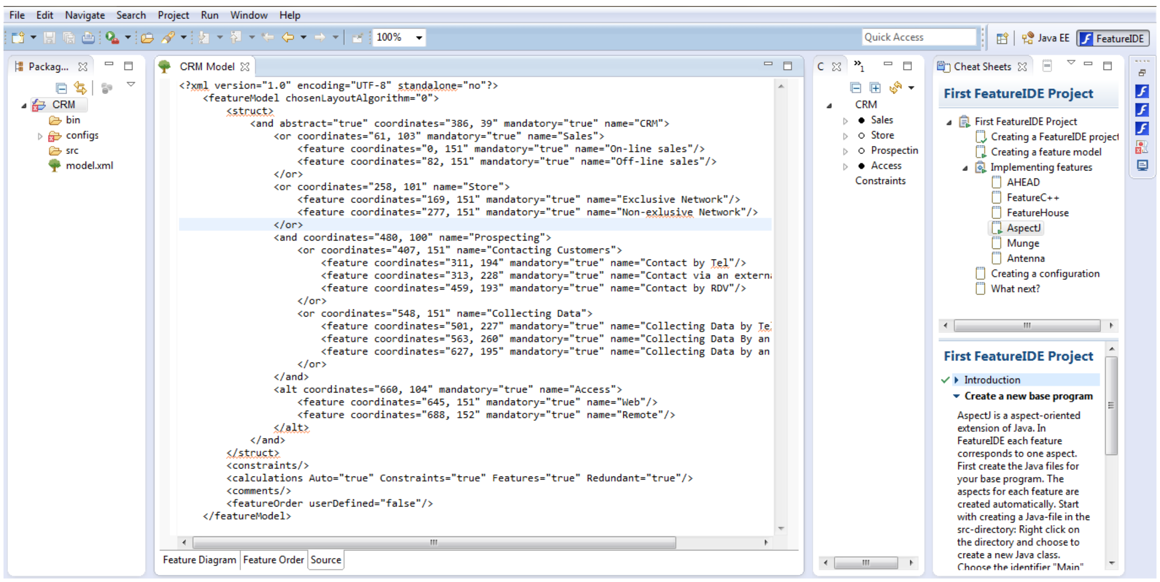
4.1. Process 1: Feature Model and Specification Transformation

During this process, the existing feature models of the SPL and the specifications of the new evolutions are transformed into the same formal representation to simplify the search of duplication between the two inputs.

4.1.1. Transforming Feature Models

In our approach, both the domain and the application models are expressed using the FODA feature model. In order to provide a formal representation of these models, we use the FeatureIDE tool [23], which is an open source framework for software product line engineering based on FOSD. This framework supports the entire life-cycle of a product line, especially domain analysis and feature modeling. Indeed, the framework provides the possibility of presenting graphically the feature model tree, allows the derivation of application models from a domain model and automatically generates the XML source of feature models.

Figure 2 depicts an example of the XML source generated by FeatureIDE from a feature model. In our approach, we suppose that the tags "and", "or" and "alt" correspond to variation points, while the tags "feature" correspond to variants.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    <featureModel chosenLayoutAlgorithm="0">
        <struct>
            <and abstract="true" coordinates="386, 39" mandatory="true" name="CRM">
                <or coordinates="61, 103" mandatory="true" name="Sales">
                    <feature coordinates="0, 151" mandatory="true" name="On-line sales"/>
                    <feature coordinates="82, 151" mandatory="true" name="Off-line sales"/>
                </or>
                <or coordinates="258, 101" name="Store">
                    <feature coordinates="169, 151" mandatory="true" name="Exclusive Network"/>
                    <feature coordinates="277, 151" mandatory="true" name="Non-exlusive Network"/>
                </or>
                <and coordinates="480, 100" name="Prospecting">
                    <or coordinates="407, 151" name="Contacting Customers">
                        <feature coordinates="311, 194" mandatory="true" name="Contact by Tel"/>
                        <feature coordinates="313, 228" mandatory="true" name="Contact via an extern
                        <feature coordinates="459, 193" mandatory="true" name="Contact by RDV"/>
                    </or>
                    <or coordinates="548, 151" name="Collecting Data">
                        <feature coordinates="501, 227" mandatory="true" name="Collecting Data by Te
                        <feature coordinates="563, 260" mandatory="true" name="Collecting Data By an
                        <feature coordinates="627, 195" mandatory="true" name="Collecting Data by an
                    </or>
                </and>
                <alt coordinates="660, 104" mandatory="true" name="Access">
                    <feature coordinates="645, 151" mandatory="true" name="Web"/>
                    <feature coordinates="688, 152" mandatory="true" name="Remote"/>
                </alt>
            </and>
        </struct>
        <constraints/>
        <calculations Auto="true" Constraints="true" Features="true" Redundant="true"/>
        <comments/>
        <featureOrder userDefined="false"/>
    </featureModel>
```

AspectJ is a aspect-oriented extension of Java. In FeatureIDE each feature corresponds to one aspect. First create the Java files for your base program. The aspects for each feature are created automatically. Start with creating a Java-file in the src-directory: Right click on the directory and choose to create a new Java class. Choose the identifier "Main"

Figure 2. The XML source of a feature model.

As a reminder, variation points are places in a design or implementation that identify the locations at which variation occurs. A variant is a single option of a variation point. As stated by Pohl *et al.* [7], a variation point must be associated with at least one variant and can offer more than one variant.

4.1.2. Transforming Natural Language Specifications

In this section, we explain the process of the transformation of natural language requirements into an XML document. To perform this operation, we will use the OpenNLP library [29], which is a machine learning-based toolkit for the processing of natural language text. Figure 3 illustrates the different steps of this process.

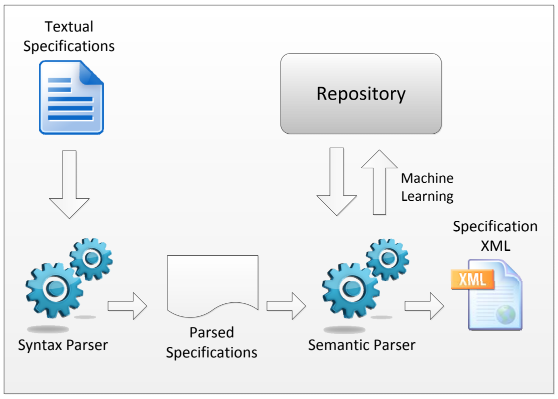


Figure 3. Transforming natural language specifications to XML.

(1) Textual Specification

The main input of this process 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. In our approach, we suppose that specifications are written in a natural language; the input of this operation can thus be a doc or a txt document.

(2) Syntactic Parser

In a previous work [30], we detailed the different actions related to the syntactic parsing of a natural language text using OpenNLP. These actions are as follows.

- Sentences detection: The first operation consists of detecting separate sentences in the specifications and writing each one in a different line.
- Tokenization: This action segments sentences into tokens. A token can be a word, punctuation, a number, *etc*. As an output of this action, all of the tokens of the specification are separated using whitespace characters, such as a space or line break, or by punctuation characters.
- Parsing: This step consists of analyzing each sentence of the specification in order to extract the words it contains and to determine their parts-of-speech based on the rules of the language grammar (e.g., noun, verb, adjective, adverb). A parser marks all of the words of a sentence using a POS tagger (part-of-speech tagger) and converts the sentence into a tree that represents the syntactic structure of the sentence. This action allows us to confirm whether the action of a verb is affirmative or negative and whether a requirement is mandatory or optional, *etc*.

(3) Repository

The repository contains two kinds of information:

- The glossary: This contains the different features used in the domain model, classified by different categories, especially "variant".
- The dictionary: This corresponds to the definition of all of the features of the product line, their synonyms and the relationships between them. This dictionary is manually built and updated during system evolutions. In our test, we do not use an existing dictionary; we have created a new one based on the domain model of the SPL. It has to be noted that the current version is not final; we are still working on it. Moreover, we intend to import existing thesauri to enrich our dictionary. Based on this dictionary, we compare the key synonyms of the new and existing variants to automatically detect duplications. Thus, a dictionary is necessary in our approach, because it is based on a semantic comparison, so we necessarily need synonyms to detect the duplications.

The content of the repository is initiated based on the domain model of the product line. So that the repository follows the evolution of the product line and its derived products, we must increment its content based on the new added features.

(4) Semantic Parser

This level involves the extraction of meaning from a sentence. For this, we feed the machine learning of OpenNLP with tagged variants from the domain model. The variants are not necessarily named entities; they can also be a part of a sentence. In order to parse the specification, we use the OpenNLP entity detector that uses a probabilistic approach to detect the potential variants that exist in the specification. This operation is automatically performed. Figure 4 presents an example of entity recognition. Once all of the variants are tagged, the corresponding tree is generated. The machine learning is updated continuously to improve the operation of variants detection and to make the initial model always up-to-date. The more the model is full with tagged variants, the more the recognition of variants from the specification is accurate. In order to measure the precision of entity recognition, we use the evaluation tool of OpenNLP, which provides information about the accuracy of the used model.

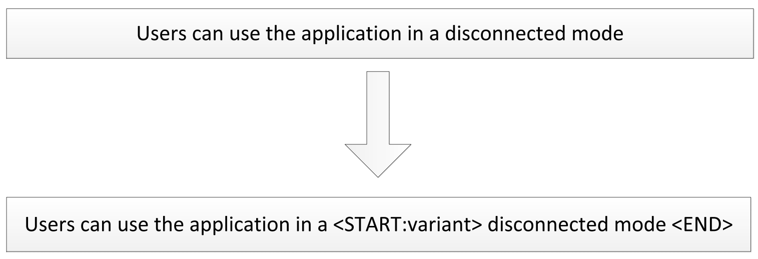

Users can use the application in a disconnected mode
Users can use the application in a <START:variant> disconnected mode <END>

Figure 4. A tagged sentence.

(5) Output

After all of the sentences of the specification are tagged, we use the different detected entities and the relationships between them to construct a graph whose structure is similar to the domain and application feature models. This graph can thus be represented in the form of an XML document and will serve as an input of the algorithm to detect duplication.

4.2. Process 2: Duplication Detection

As we stated in [31], to detect duplication introduced by new specifications into the existing feature models, we first need to be sure that the specifications and the feature models are duplication free. Thus, the detection of duplication must be performed in three main steps:

- Detecting duplication in the new specifications
- Detecting duplication in the existing feature models
- Detecting duplication between the specifications and the feature models

For each step, we propose one or several algorithms to carry out the required operation. In the remainder of this section, we present some essential predicates related to our solution. Then, we explain the different algorithms proposed to deal with duplication detection.

4.2.1. Formalizing the Basic Concepts

Before presenting the different algorithms, we need to formalize the basic concepts used in our solution.

- Formalizing the domain model

We denote by D the domain feature model of the product line. PD is the set of nodes related to the tags "and", "or" and "alt" of D (*i.e.*, variation points), and VD is the set of nodes related to the tags "feature" of D (*i.e.*, variants).

$$PD = \{pd_1, pd_2, \ldots, pd_m\}$$

$$\forall pd_i \in PD \quad \exists VD_i \text{ where } VD_i = \{vd_{ij} \quad | \quad j \in \mathbb{N}\}$$

Thus:

$$VD = \bigcup_{i=1}^{m} VD_i$$

This definition involves two assumptions:

- A variation point pd_i from the set PD is associated with a set of variants VD_i that contains the elements vd_{ij}.
- VD is the union of all of the sets of variants VD_i associated with the variation points. It represents the set of all of the variants of the domain model.

The verification of duplication is carried out for all of the variants related to variation points, which means the set of pairs (pd_i, vd_{ij}).

- Formalizing the application model

We denote by A the application feature model of a derived application. PA is the set of variation points of A, and VA is the set of variants of A.

$$PA = \{pa_1, pa_2, \ldots, pa_m\}$$

$$\forall pa_i \in PA \quad \exists VA_i \text{ where } VA_i = \{va_{ij} \quad | \quad j \in \mathbb{N}\}$$

Thus:

$$VA = \bigcup_{i=1}^{m} VA_i$$

- Formalizing the specification

Similarly, we denote by S the specification of a new evolution. P is the set of variation points of S, and V is the set of variants of S.

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

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

Thus:

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

It has to be noted that a variation point must be associated with at least one variant. This assumption is required by Pohl *et al.* [7] when defining the variability of an SPL.

4.2.2. The Algorithms of Duplication Detection

The verification of duplication is carried out through a set of algorithms detailed in what follows.

- Detecting Duplication in Specifications

The algorithm responsible for detecting duplication in specifications was presented in [30]. The algorithm contains the four steps depicted in Table 2.

Table 2. The algorithm of duplication detection in specifications.

Step	**Actions**
Step 1	Definition of a key synonym for each set of synonyms, based on the dictionary. 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". Generation of an equivalent XML, 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.
Step 4	Comparison between the variants of all of the variation points, in order to detect duplication in the whole XML.

- Detecting Duplication in Feature Models

Since the representation of feature models and specifications is unified during the first process of the framework, the algorithm to detect duplication in feature models is the same as the one used with specifications. It has to be noted that this algorithm must be applied both to domain and application models of the product line.

- Detecting Duplication between Specifications and Feature Models

The domain and application models of a software product line generally co-evolve independently of each other [10]. Thus, the verification of duplication is carried out through two algorithms. The first algorithm involves the comparison between the domain model and the specification, while the second algorithm concerns the comparison between the application model and the specification. The two algorithms are similar, but we chose to do the two verifications because the decision will change. Indeed, if a duplication is detected between the specification and the application model, no actions are required. However, if a duplication exists between the specification and the domain model, the feature must be derived in the specific evolving product.

Algorithm 1 is the algorithm that corresponds to the second verification. The algorithm consists of comparing each variation point of the specification with the variation points of the application feature model. When an equivalent is found in the latter, the algorithm starts another comparison between the variants corresponding to the variation point of the specification and the variants related to the equivalent variation point of the application model. If a variant is detected, this means that the feature corresponding to the pair (variation point, variant) is duplicated. In the case that the new variants related to a variation point do not have equivalents in the application model, no duplication is detected, and the variants can be safely added to the existing model.

Algorithm 1 Detecting duplication between the specification and the application model.

```
Principal Lookup :
for each $p_i \in P$ do
    for each $pa_k \in PA$ do
        if $Equiv(p_i, pa_k)$ then
            Secondary Lookup :
            for each $v_{ij} \in V_i$ do
                for each $va_{kl} \in VA_k$ do
                    if $Equiv(v_{ij}, v_{kl})$ then
                        $NoticeDuplication(p_i, v_{ij}, pa_k, v_{kl})$
                        $Continue$ Secondary Lookup
                    end if
                end for
            end for
            $Continue$ Principal Lookup
        end if
    end for
end for
```

4.2.3. The Framework Outputs

For each algorithm, the duplicated features are stored in a log file that will be sent to the user in order to inform him of the duplication and its location and to invite him to make his decisions regarding the features (e.g., modify the new feature, delete the new/old feature, replace the old feature).

4.3. Automated Tool

For the aim of implementing the proposed framework, we have started developing a support tool that automatizes the framework processes. The technologies used in the development process are:

- Development environment: Eclipse
- Development language: Java
- Human-computer interface: SWT
- Database: PostGre SQL
- Feature models creation: FeatureIDE
- Natural language processing: OpenNLP
- Graph generation: Prefuse

The functionalities that the automated tool must provide are:

- The processing of natural language specifications (a txt document) and their transformation to an XML document.
- The creation and modification of the repository.
- The detection of duplication in specifications.
- The detection of duplication in the XML source of a feature model.
- The detection of duplication between a feature model and a specification.

5. Results and Discussion

To illustrate our solution, we use a CRM (customer relationship management) product line with the feature model depicted in Figure 5.

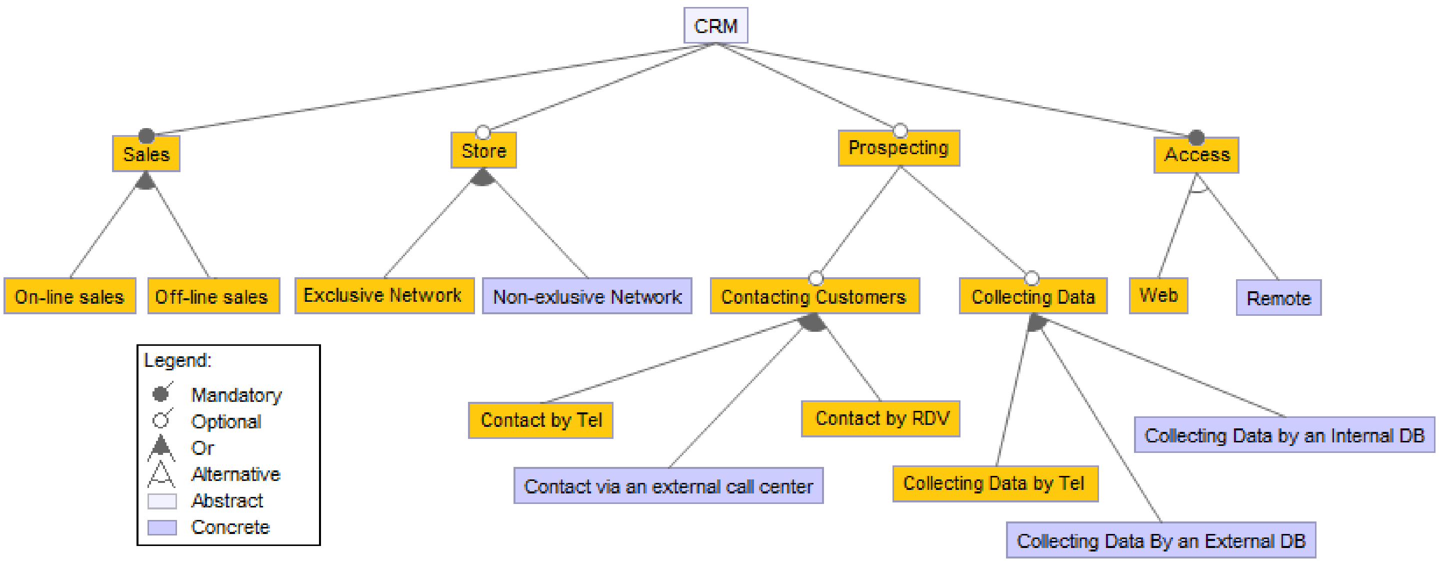


Figure 5. The domain feature model of the CRM.

For our test, we consider a derived application of the CRM. It is a web application that enables the management of on-line and off-line sales. The kind of stores managed by the application are the exclusive network stores only. The sector header collects information about customers by telephone and contacts customers by telephone or by making an appointment. In Figure 5, the features related to this specific application are presented in orange.

It has to be noted that the features of this CRM are initially used to populate the repository. The main interface of the developed tool is depicted in Figure 6. It contains a menu with two entries. The first entry enables the user to upload the feature model or the specification, while the second enables the visualization of the repository content.

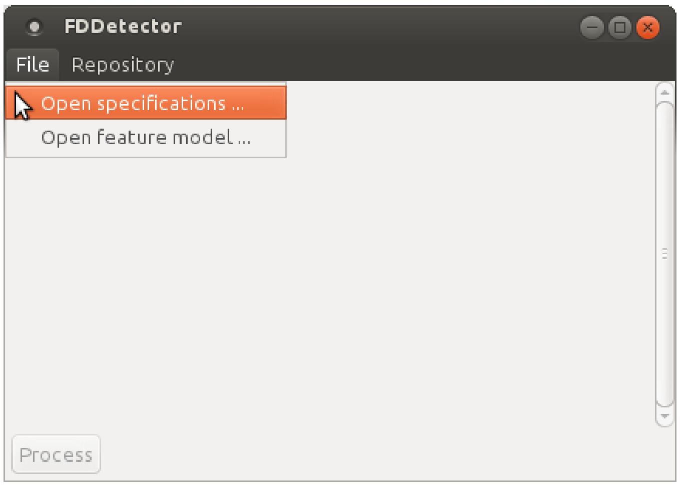
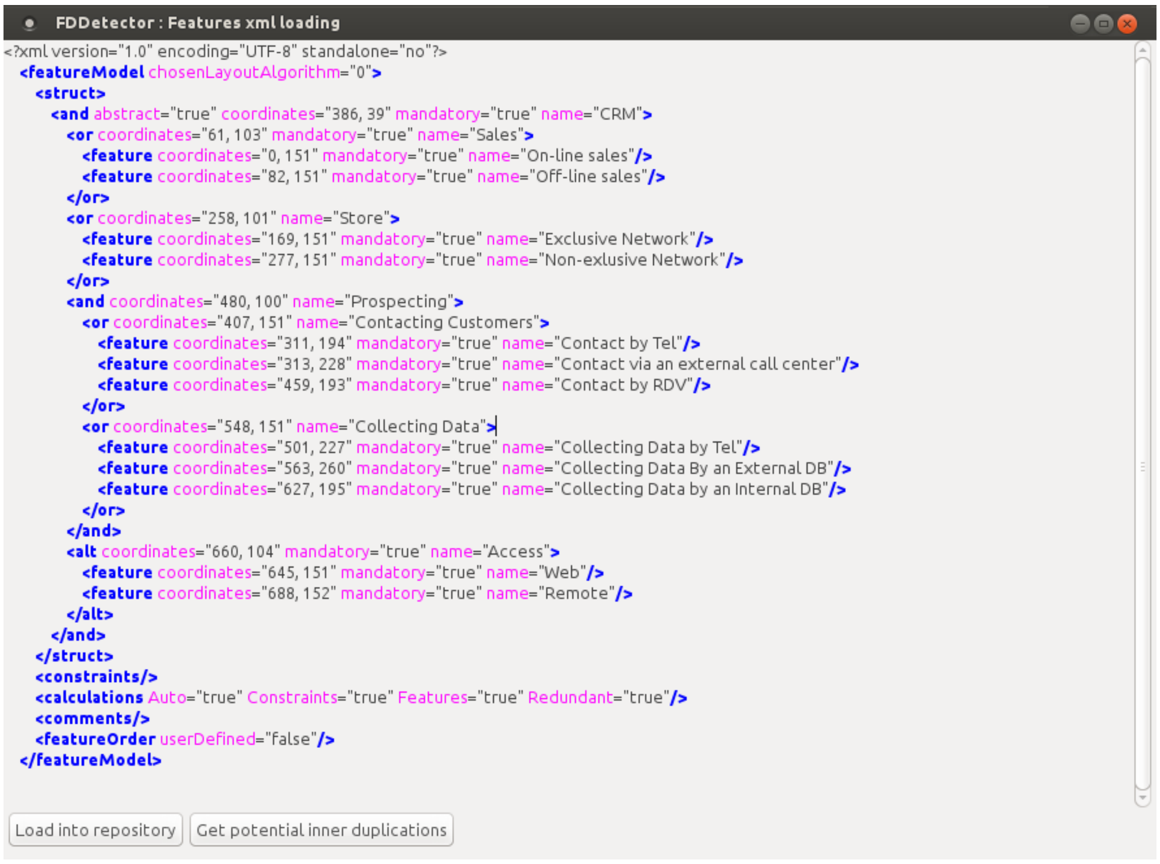


Figure 6. The main interface of the support tool.

To load the domain and application models in the repository, we use the button "Open Feature Model" of the "File" entry. Figure 7 presents an uploaded feature model.

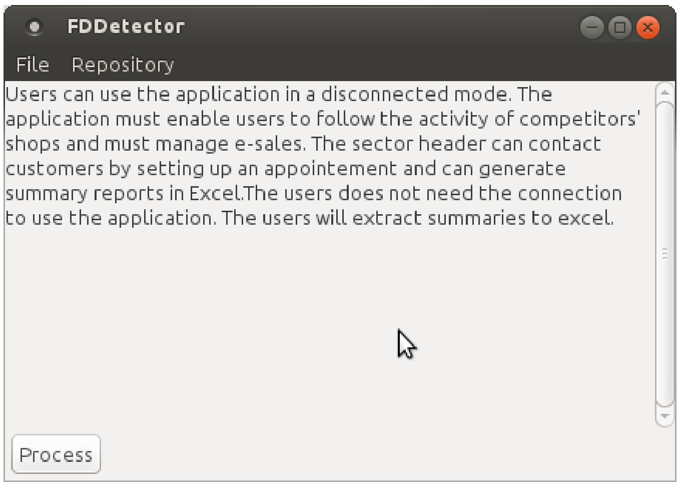

FDDetector : Features xml loading
Load into repository
Get potential inner duplications

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<featureModel chosenLayoutAlgorithm="0">
  <struct>
    <and abstract="true" coordinates="386, 39" mandatory="true" name="CRM">
      <or coordinates="61, 103" mandatory="true" name="Sales">
        <feature coordinates="0, 151" mandatory="true" name="On-line sales"/>
        <feature coordinates="82, 151" mandatory="true" name="Off-line sales"/>
      </or>
      <or coordinates="258, 101" name="Store">
        <feature coordinates="169, 151" mandatory="true" name="Exclusive Network"/>
        <feature coordinates="277, 151" mandatory="true" name="Non-exlusive Network"/>
      </or>
      <and coordinates="480, 100" name="Prospecting">
        <or coordinates="407, 151" name="Contacting Customers">
          <feature coordinates="311, 194" mandatory="true" name="Contact by Tel"/>
          <feature coordinates="313, 228" mandatory="true" name="Contact via an external call center"/>
          <feature coordinates="459, 193" mandatory="true" name="Contact by RDV"/>
        </or>
        <or coordinates="548, 151" name="Collecting Data">
          <feature coordinates="501, 227" mandatory="true" name="Collecting Data by Tel"/>
          <feature coordinates="563, 260" mandatory="true" name="Collecting Data By an External DB"/>
          <feature coordinates="627, 195" mandatory="true" name="Collecting Data by an Internal DB"/>
        </or>
      </and>
      <alt coordinates="660, 104" mandatory="true" name="Access">
        <feature coordinates="645, 151" mandatory="true" name="Web"/>
        <feature coordinates="688, 152" mandatory="true" name="Remote"/>
      </alt>
    </and>
  </struct>
  <constraints/>
  <calculations Auto="true" Constraints="true" Features="true" Redundant="true"/>
  <comments/>
  <featureOrder userDefined="false"/>
</featureModel>
```

Figure 7. The feature model loading.

This interface gives the possibility to load the model content into the repository. In addition, it enables the detection of duplications inside this feature model.

The second input of our test is the specification of a new evolution of the derived application. We added intentionally two duplicate features in the specification in order to verify whether our tool will detect them or not. The specification is illustrated in Figure 8.



Users can use the application in a disconnected mode. The application must enable users to follow the activity of competitors' shops and must manage e-sales. The sector header can contact customers by setting up an appointement and can generate summary reports in Excel.The users does not need the connection to use the application. The users will extract summaries to excel.

Figure 8. The specification of the new evolution.

When we press the button "Process", the specification is processed, transformed to an XML document, and the algorithm of duplication detection is applied. Figure 9 presents the graphic form of the generated XML. This presentation facilitates the visualization of the different new features introduced by the specification and enables the distinction of duplicate features by presenting them in a different color. For our test, the program detected two duplications as displayed in the top left corner of the interface. Moreover, the graph also distinguishes the new variants added by the specification by relating them to the node "unbound variants". The user will be able to bind these variants with a variation point from the repository.

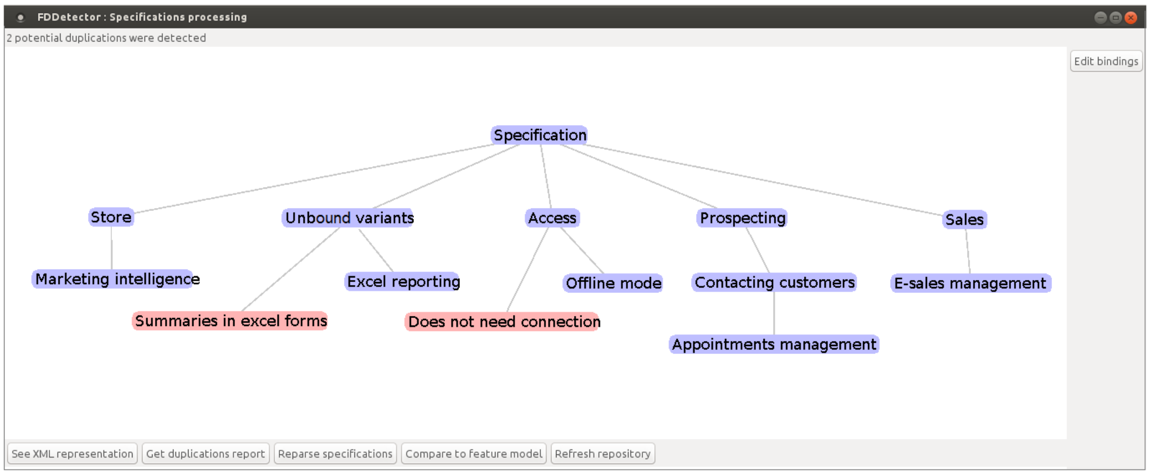
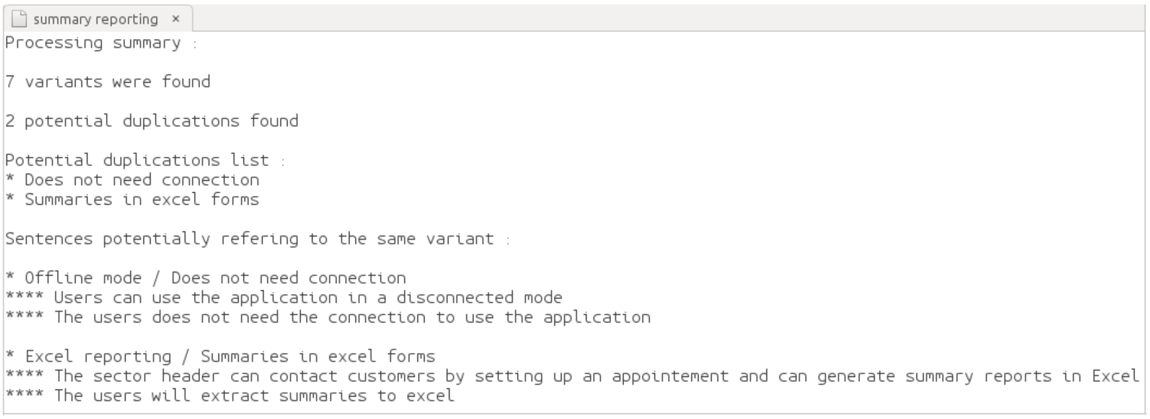


Figure 9. The graph corresponding to the specification.

This interface provides other functionalities, such as the display of the generated XML, the re-parsing of the specification after a modification of the repository, the comparison between the specification and the feature model to detect duplication between them, the refreshment of the repository after the binding of new features and the generation of the log. The log related to our test is presented in Figure 10.



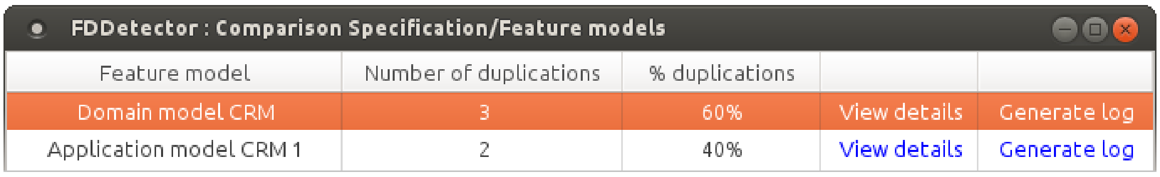
summary reporting ×

```
Processing summary :

7 variants were found

2 potential duplications found

Potential duplications list :
* Does not need connection
* Summaries in excel forms

Sentences potentially refering to the same variant :

* Offline mode / Does not need connection
**** Users can use the application in a disconnected mode
**** The users does not need the connection to use the application

* Excel reporting / Summaries in excel forms
**** The sector header can contact customers by setting up an appointement and can generate summary reports in Excel
**** The users will extract summaries to excel
```

Figure 10. The log of duplication detection in specifications.

This document contains the number of variants in the specification, the number of potential duplications. It also provides the sentences of the specification that contain duplication.

To search duplications between the specification and the feature models, we use the button "Compare to feature models". The result of this operation is illustrated in Figure 11. Three duplications are detected in the domain model, and two were detected in the application model. The feature that is not duplicated in the application model can be derived from the domain model. The percentages calculated in this interface will enable us to estimate the gain in the development cost.



FDDetector : Comparison Specification/Feature models

Feature model	Number of duplications	% duplications		
Domain model CRM	3	60%	View details	Generate log
Application model CRM 1	2	40%	View details	Generate log

Figure 11. The results of the comparison between the specification and feature models.

Based on the results, we deduce that our solution is capable of detecting duplication in a specification and between a feature model and a specification. However, the test was performed on a product line with a limited number of features. In our future work, we intend to carry out further evaluation through large-scale and real-life product lines to ensure the efficacy of our solution. Moreover, we will also measure the impact of duplication detection on the evolution of software product lines, by calculating the effort of the implementation of the automated tool and comparing it to the effort of implementation of new evolutions of a product line. This evaluation will enable us to estimate after how many evolutions our solution becomes cost-effective.

6. Related Work

In this section, we provide an overview of the studies most relevant to our work by classifying them into three categories.

6.1. Detecting Defects in Specifications

For the aim of automatically validating and correcting textual requirements, Holtmann *et al.* [25] 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 requirement metamodel. Then, structural patterns are specified based on this metamodel. The use of patterns allows an automated correction of some requirement 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. Specifically, we chose XML because it is the format most used in FOSD projects to represent feature models. Lami *et al.* [15] present a methodology and a tool called QuARS (Quality Analyzer for Requirement Specifications), which performs an initial parsing of the specifications in order to detect specific linguistic defects automatically, namely inconsistency, incompleteness and ambiguity. Although this study converges with our paper in the fact that it is based on parsing of natural language requirements to detect defects, QuARS is limited to syntax-related issues of a natural language document, while our approach focuses on a semantic problem, which is feature duplication. Ambriola and Gervasi [32] introduced a system called CIRCE that uses natural language processing to extract information from natural language requirements, allows one to check and measure the consistency of requirements and produces functional metric reports. Our approach is different because, on the one hand, it focuses on semantic duplication in SPLs and not on inconsistency in general, and on the other hand, because our goal is not only to detect duplication in textual requirements, but between the specifications and the existing feature models of an evolving SPL.

6.2. Detecting Defects in Feature Models

Several papers in the literature have addressed model defects caused by SPL evolution. For example, Guo and Wang [33] proposes to limit the consistency maintenance to the part of the feature model that is affected by the requested change instead of the whole feature model. In order to locate inconsistency in the domain feature model of an SPL, Yu [34] 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 of the AFTMs to extract the domain feature tree model (DFTM), which enables one to figure out the traceability between domain requirements and DFTM. Using this method helps with automatically constructing 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, because we suppose that the domain and application models exist; our objective is hence to construct a more formal presentation of them to facilitate the search of the new features in these models. In [16], Mazo provides a classification of different verification criteria of the product line model that he categorizes into four families: expressiveness criteria, consistency criteria, error-prone criteria and redundancy-free criteria. Redundancy can easily be confused with duplication, but it is completely different, because Mazo focuses on redundancy of dependencies and not redundancy of features. The same study defines also different conformance checking criteria, among which two features should not have the same name in the same model. This is also different from our approach, which is based on semantic equivalence and not only on the equality of features.

6.3. Detecting Duplication in SPLs

Software cloning or duplication has been actively addressed recently, especially in software product lines. For example, Schulze [35] conducted an empirical study that emphasizes the existence of clones in SPLs and identifies the different causes behind them. Then, he proposes variant-preserving refactoring to remove code clones from compositional software product lines. Similarly, Dubinsky *et al.* [36] conducted an empirical study to investigate the cloning culture in six industrial software product lines. Based on the results of this study, the authors aim at developing better methodologies and tools that promote efficient software reuse. Mende *et al.* [37] provide a tool support for the grow-and-prune model in the evolution of software product lines by detecting pairs of functions sharing code, then by measuring the textual similarity among these functions. The common denominator between all of these approaches is that they focus on duplication at the code level, while our approach proposes a solution to duplication at the feature level, which enables one to avoid duplication-related problems from the very beginning.

7. Conclusions and Future Work

Many papers in literature have addressed the detection and correction of model defects caused by SPL evolution, but the majority of them focused on inconsistency in general, while duplication of features has not been thoroughly discussed. The aim of this paper was to present a novel approach for identifying duplicate features when evolving a derived product of an SPL. Since specifications received from customers are most of the time expressed in natural language, we based our approach on natural language processing techniques. We thus proposed a two-process framework. The first process consists of transforming the model domain, the application domain and the specifications into a unified form, XML trees. The second process contains a set of algorithms that detect duplication in specifications, in feature models and between specifications and feature models. In order to automatize the two processes, a support tool was proposed. So far, we applied the algorithm of duplication detection to a simple CRM tool to verify its efficacy. In a future work, we intend to apply our solution to a large-scale product line. In addition, we will carry out a quantitative evaluation to estimate the effort gained by using the proposed method.

Author Contributions

The article was written by Amal Khtira. The research and state of the art were done by Amal Khtira and Anissa Benlarabi. The design and implementation of the solution were done by Amal Khtira and Bouchra El Asri. All authors have read and approved the final manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Clements, P.; Northop, L. *Software Product Lines—Practices and Patterns*; Addison-Wesley: Boston, MA, USA, 2002.
2. Apel, S.; Kästner, C. An overview of feature-oriented software development. *J. Object Technol.* **2009**, *8*, 49–84.
3. Kang, K.; Cohen, S.; Hess, J.; Novak, W.; Peterson, S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*; Technical Report CMU/SEI-90-TR-21; Software Engineering Institute: Pittsburgh, PA, USA, 1990.
4. Romero, D.; Urli, S.; Quinton, C.; Blay-Fornarino, M.; Collet, P.; Duchien, L.; Mosser, S. SPLEMMA: A generic framework for controlled-evolution of software product lines. In Proceedings of the 17th International Software Product Line Conference Co-located Workshops, Tokyo, Japan, 26–30 August 2013; pp. 59–66.
5. Zowghi, D.; Gervasi, V. On the interplay between consistency, completeness, and correctness in requirements evolution. *Inf. Softw. Technol.* **2004**, *46*, 763–779.
6. Khtira, A.; Benlarabi, A.; El Asri, B. Towards duplication-free feature models when evolving software product lines. In Proceedings of the 9th International Conference on Software Engineering Advances, Nice, France, 12–16 October 2014; pp. 107–113.
7. Pohl, K.; Böckle, G.; van der Linden, F. *Software Product Line Engineering Foundations, Principles, and Techniques*; Springer: Berlin, Germany, 2005.
8. Anquetil, N.; Kulesza, U.; Mitschke, R.; Moreira, A.; Royer, J.C.; Rummler, A.; Sousa, A. A model-driven traceability framework for software product lines. *Softw. Syst. Model.* **2010**, *9*, 427–451.
9. Pleuss, A.; Botterweck, G.; Dhungana, D.; Polzer, A.; Kowalewski, S. Model-driven support for product line evolution on feature level. *J. Syst. Softw.* **2010**, *85*, 2261–2274.
10. Benlarabi, A.; El Asri, B.; Khtira, A. A co-evolution model for software product lines: An approach based on evolutionary trees. In Proceedins of the IEEE Second World Conference on Complex Systems, Agadir, Morocco, 10–12 November 2014; pp. 140–145.
11. Seidl, C.; Heidenreich, F.; Assmann, U. Co-evolution of models and feature mapping in software product lines. In Proceedings of the 16th International Software Product Line Conference, Salvador, Brazil, 2–7 September 2012; Volume 1, pp. 76–85.
12. Tizzei, L.P.; Dias, M.; Rubira; C.M.; Garcia, A.; Lee, J. Components meet aspects: Assessing design stability of a software product line. *Inf. Softw. Technol.* **2011**, *53*, 121–136.
13. Nuseibeh, B. To be and not to be: On managing inconsistency in software development. In Proceedings of the 8th International Workshop on Software Specification and Design, Schloss Velen, Germany, 22–23 March 1996; pp. 164–169.
14. Easterbrook, S.; Nuseibeh, B. Managing inconsistencies in an evolving specification. In Proceedings of the IEEE Second IEEE International Symposium on Requirements Engineering, York, UK, 27–29 March 1995; pp. 48–55.
15. Lami, G.; Gnesi, S.; Fabbrini, F.; Fusani, M.; Trentanni, G. An automatic tool for the analysis of natural language requirements. *Comput. Syst. Eng.* **2005**, *20*, 53–62.

16. Mazo, R. A Generic Approach for Automated Verification of Product Line Models. Ph.D. Thesis, Pantheon-Sorbonne University, Paris, France, 2011.
17. Hunt, A.; Thomas, D. *The Pragmatic Programmer: From Journeyman to Master*; Addison-Wesley: Boston, MA, USA, 2000.
18. Neves, L.; Teixeira, L.; Sena, D.; Alves, V.; Kulezsa, U.; Borba, P. Investigating the safe evolution of software product lines. *ACM SIGPLAN Not.* **2012**, *47*, 33–42.
19. Aversano, L.; Cerulo, L.; Di Penta, M. How clones are maintained: An empirical study. In Proceedings of the IEEE 11th European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, 21–23 March 2007; pp. 81–90.
20. Salinesi, C.; Mazo, R.; Djebbi, O.; Diaz, D.; Lora-Michiels, A. Constraints: The core of product line engineering. In Proceedings of the IEEE 5th International Conference on Research Challenges in Information Science, Gosier, France, 19–21 May 2011; pp. 1–10.
21. Kang, K.C.; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.* **1998**, *5*, 143–168.
22. Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*; ACM Press: New York, NY, USA; Addison-Wesley: New York, NY, USA, 2000.
23. Kästner, C.; Thüm, T.; Saake, G.; Feigenspan, J.; Leich, T.; Wielgorz, F.; Apel, S. FeatureIDE: A tool framework for feature-oriented software development. In Proceedings of the IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–27 May 2009; pp. 611–614.
24. Meyer, B. On formalism in specifications. *IEEE softw.* **1985**, *2*, 6–26.
25. Holtmann, J.; Meyer, J.; von Detten, M. Automatic validation and correction of formalized, textual requirements. In Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, Berlin, Germany, 21–25 March 2011; pp. 486–495.
26. Fatwanto, A. Software requirements specification analysis using natural language processing technique. In Proceedings of the IEEE International Conference on QiR (Quality in Research), Yogyakarta, Indonesia, 25–28 June 2013; pp. 105–110.
27. Ilieva, M.G.; Ormandjieva, O. Automatic transition of natural language software requirements specification into formal presentation. In *Natural Language Processing and Information Systems*; Springer: Berlin, Germany, 2005; pp. 392–397.
28. Khtira, A.; Benlarabi, A.; El Asri, B. An approach to detect duplication in software product lines using natural language processing. In Proceedings of the Mediterranean Conference on Information and Communication Technologies, Saidia, Morocco, 7–9 May 2015; in press.
29. The Apache Software Foundation. OpenNLP. Available online: http://opennlp.apache.org/ (accessed on 5 October 2015).
30. Khtira, A.; Benlarabi, A.; El Asri, B. Detecting feature duplication in natural language specifications when evolving software product lines. In Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, 29–30 April 2015.

31. Khtira, A. Towards a framework for feature deduplication during software product lines evolution. In Proceedings of the 27th International Conference on Advanced Information Systems Engineering, Stockholm, Sweden, 8–12 June 2015.
32. Ambriola, V.; Gervasi, V. Processing natural language requirements. In Proceedings of the 12th IEEE International Conference on Automated Software Engineering, Incline Village, NV, USA, 1–5 November 1997; pp. 36–45.
33. Guo, J.; Wang, Y. Towards consistent evolution of feature models. In *Software Product Lines: Going Beyond*; Springer: Berlin, Germany, 2010; pp. 451–455.
34. Yu, D.; Geng, P.; Wu, W. Constructing traceability between features and requirements for software product line engineering. In Proceedings of the 19th Asia-Pacific Software Engineering Conference, Hong Kong, China, 4–7 December 2012; Volume 2, pp. 27–34.
35. Schulze, S. Analysis and Removal of Code Clones in Software Product Lines. Ph.D. Thesis, Magdeburg University, Magdeburg, Germany, 2012.
36. Dubinsky, Y.; Rubin, J.; Berger, T.; Duszynski, S.; Becker, M.; Czarnecki, K. An exploratory study of cloning in industrial software product lines. In Proceedings of the IEEE 17th European Conference on Software Maintenance and Reengineering, Genova, Switzerland, 5–8 March 2013; pp. 25–34.
37. Mende, T.; Beckwermert, F.; Koschke, R.; Meier, G. Supporting the grow-and-prune model in software product lines evolution using clone detection. In Proceedings of the IEEE 12th European Conference on Software Maintenance and Reengineering, Athens, Greece, 1–4 April 2008; pp. 163–172.