A Framework for Ensuring Non-Duplication of Features in Software Product Lines

Amal Khtira[1], Anissa Benlarabi[2], Bouchra El Asri[3]
IMS Team, SIME Laboratory, ENSIAS, Mohammed V University
Rabat, Morocco
[1]amalkhtira@gmail.com, [2]a.benlarabi@gmail.com, [3]elasri@ensias.ma

***Abstract*—Since the emergence of Software Product Line Engineering, the requirements evolution issue has been addressed by many researchers and many approaches have been proposed. However, most studies focused on evolution in domain engineering while application engineering has not received the same attention. During the evolution of a derived product, new features are added or modified in the application model, which may cause many model defects, such as inconsistency and duplication. These defects are introduced to the existing models from the non-verified specifications related to the SPL evolutions. Since these specifications are most of the time expressed in natural language, the task of detecting defects becomes more complicated and error-prone. The aim of this paper is to present a framework that transforms both the SPL feature models and the specification of a new evolution into a more formal representation and provides algorithms to determine the duplicated features between the specification and the existing models. In addition, we describe a support tool created based on the framework and we evaluate the efficacy of our approach using an open source product line.**

***Keywords*—Software Product Line Evolution; Domain Engineering; Application Engineering; Feature Duplication; Natural Language Processing.**

I. INTRODUCTION

Feature duplication, as described in [1], occurs when two or more features of the same semantics exist in a feature model of a SPL. SPLs, contrarily to single software, have emerged as a solution to develop different applications based on a core platform. The adoption of SPLs by companies enables them to reduce time to market, to reduce cost and to produce high quality applications. Another major advantage of SPLs is the reuse of core assets to generate specific applications according to the need of customers.

The Software Product Line Engineering (SPLE) approach consists of two processes, namely, domain engineering and application engineering [2]. During these processes, a number of artefacts are produced which encompass requirements, architecture, components and tests. Domain engineering involves identifying the common and distinct features of all the product line members, creating the design of the system and implementing the reusable components. During application engineering, individual products are derived based on the artefacts of the first process, using some techniques of derivation.

Many issues related to SPLE have been addressed both by researchers and practitioners, such as reusability, product derivation, variability management, etc. The focus of our study will be on SPL evolution. Evolution is defined by Madhavji et al. [3] as "*a process of progressive change and cyclic adaptation over time in terms of the attributes, behavioral properties and relational configuration of some material, abstract, natural or artificial entity or system*". This definition applies to different domains, including software engineering.

In the literature, several studies have dealt with evolution in Software Product Lines (SPLs). Xue et al. [4] presented a method to detect changes that occurred to product features in a family of product variants. In order to support agile SPL evolution, Urli et al. [5] introduces the Composite Feature Model (CFM), which consists of creating small Feature Models (FMs) that corresponds each to a precise domain. Other approaches, such as Ahmad et al.'s [6], focused on the extraction of architecture knowledge in order to assess the evolutionary capabilities of a system and to estimate the cost of evolution. Some papers focused on the co-evolution of different elements of SPLs [7].

Based on the literature, we have found that most of the studies addressing software evolution focus on domain engineering, while application engineering has not received the same interest. However, the experience has proven in many industrial contexts that systems continue to change even after the product derivation. This change can be the source of many problems in the product line such as inconsistency and duplication. Indeed, the core assets of the product line and the artefacts of derived products are most of the time maintained by different teams. Moreover, developers under time pressure can forget to refer to the domain model before starting to implement the changes. For these reasons and others, duplication in SPL can easily happen. Since the requirements related to SPL evolutions are most of the time expressed in the form of natural language specifications, the task of model verification becomes difficult and error-prone. In order to simplify the detection of defects, many studies have proposed methods to transform natural language specifications into formal or semi-formal specifications [8][9][10].

In this paper, we describe our framework that consists of unifying the representation of the natural language specifications and the existing domain and application models, then detecting duplicate features using a set of algorithms. We extend the work presented in [1] by:

- Completing the background of our work by adding an overview of the presentation of specifications related to SPL evolutions.
- Presenting an improved version of the proposed framework and describing its different processes in details.
- Enhancing the formalization of the framework concepts.

- Providing the pseudo-code of the algorithm of duplication detection.
- Describing the architecture and main functionality of a support tool created to evaluate the framework.
- Applying the proposed solution to an open source case study.

The remainder of the paper is structured as follows. Section II gives an overview of the background of our study and describes the problem we are dealing with. In Section III, we present the basic concepts and the overview of the proposed framework. In Section IV, we provide a formalization of the basic concepts before describing the algorithm of deduplication. Section V presents the architecture and the main features of the automated tool created based on the framework. An application of the framework on a case study is presented in Section VI. Section VII positions our approach with related works. The paper is concluded in Section VIII.

II. Background and Objective

In this section, we introduce the background of our study. First, we present the SPLE paradigm. Then, we highlight the problem of duplication when evolving products in application engineering, and finally, we give an insight of the presentation of specifications related to SPLs, before explaining the objective of our work.

A. Software Product Line Engineering

A SPL is defined by Clements and Northop [11] 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 main goals of a SPL are to reduce the cost of developing software products, to enhance quality and to promote reusability.

The domain engineering phase of the SPLE framework is responsible for defining the commonality and variability of the applications of the product line. Capturing the common features of all the applications increases the reusability of the system, and determining the variant features allows the production of a large number of specific applications that satisfy different needs of customers. When the domain model is ready, the next step consists of creating the design of the system that contains the software components and their relationships. Those components are then implemented and the code of the product line is generated.

The process of creating a specific product based on a SPL is referred to as product derivation or product instantiation. Product derivation consists of taking a snapshot of the product line by binding variability already defined in the domain engineering and using it as a starting point to develop an individual product. This process is applied during application engineering phase and is responsible for instantiating all the artefacts of the product line, i.e., model, design, components, etc.

In order to document and model variability in SPL, many approaches have been proposed. For instance, Pohl et al. [2] proposed the orthogonal variability model to define the variability in a dedicated model, while other papers preferred to integrate the variability in the existing models, such as UML models or FODA models [12]. Another approach proposed by Salinesi et al. [13] used a constraint-based product line language. In our approach, we will use the FODA method.

B. Duplication of Features during SPL Evolution

The goal of SPLE is to make an up-front investment to create the platform. Indeed, during domain engineering, the requirements of all the potential applications are captured, and as far as possible, the scenarios of the possible changes have to be predicted and anticipated. The evolution and maintenance of the product line are conducted through several iterations until the platform becomes as stable as possible. As new evolutions arise, the domain artefacts are adapted and refined.

On the one hand, the team responsible for developing and maintaining the product line studies the requirements of each customer and derives specific applications that respond to these requirements. On the other hand, a different team takes in charge the maintenance of each application. Following the logic of SPLE, the derived applications are not supposed to change much, but the experience has shown that this assumption is not always true. In fact, even after the derivation of a specific product, new demands can be received from the customer, either changes to existing features or addition of new ones.

During the maintenance of a product, duplication of knowledge can easily happen when evolving the model, the design or the code. In [14], four categories of duplication are distinguished:

- Imposed duplication: Developers cannot avoid duplication because the technology or the environment seems to impose it.
- Inadvertent duplication: This type of duplication comes about as a result of mistakes in the design. In this case, the developers are not aware of the duplication.
- Impatient duplication: When the time is pressing and deadlines are looming, developers get impatient and tend to take shortcuts by implementing as quick as possible the requirements of customers. In these conditions, duplication is very likely to happen.
- Inter-developer duplication: Different people working on one product can easily duplicate information.

In the context of SPLE, at least the three last categories might occur. Indeed, when a derived application is shipped, developers responsible for maintaining it do not have a clear visibility of the domain model because another team conceived it. Thus, developers of the application may add features which are already satisfied in the domain model and have only to be derived or configured. In addition, under time pressure, developers do not refer to the application model and might add features which are already implemented.

C. Specifications of SPL Evolutions

In software engineering projects, specifications are an important input of the requirements analysis. These specifications

are most of the time written in natural language. Indeed, this communication channel is obviously what customers often use to express easily what they expect from the system and to explain their perception of the problem. However, specifying requirements in a natural language frequently makes them prone to many defects. In [15], Meyer details seven problems with natural language specifications: noise, silence, overspecification, contradictions, ambiguity, forward references and wishful thinking. Lami et al. [16] focused on other defects, namely the ambiguity, the inconsistency and the incompleteness. In order to deal with these problems, several methods have been proposed in the literature to transform natural language specifications to formal or semi-formal specifications [8][9][10].

In the case 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 the new evolutions concerning a derived product can be expressed using natural language. In order to avoid the introduction of defects (Duplication in our case) into the existing feature models, we need to formalize the presentation of these specifications, which facilitates their verification against the SPL models.

D. Objective

To the best of our knowledge, few attempts have dealt with feature duplication when evolving derived products of SPLs. Hence, the aim and contribution of this paper is to provide a framework that helps developers avoid duplication in a SPL when evolving a specific product. To this end, we will first unify the presentation of the domain and application feature models of the product line and the natural language specifications of an evolution related to a derived product. Then, an algorithm is proposed to detect the duplicated features between these two inputs.

III. A Framework to Avoid Duplication when Evolving Derived Products

In this section, we first provide a short definition of the basic concepts used in the framework, then we present the overview of the framework.

A. Basic Concepts

Before going any further, we will give an insight of the basic concepts used in the framework.

Requirement: According to [17], a Requirement is:

1) A condition or capability needed by a user to solve a problem or achieve an objective.
2) A condition or capability that must or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3) A documented representation of a condition or capability as in (1) or (2).

Domain Model: A domain is a family of related products, and the domain model is the representation of all the different and common features of these products. There are many types of domain models, but the most interesting are the feature model [12] and the variability model [2].

Application Model: The model corresponding to an individual application. It is generated by binding the variability of the domain model in a way that satisfies the needs of a specific customer [2].

Feature: A feature is the abstraction of functional or non-functional requirements that help characterize the system and must be implemented, tested, delivered, and maintained [18][19].

Feature Model: It is the description of all the possible features of a software product line and the relationships between them. The most common representation of feature models is by using the FODA feature diagram proposed by [12]. The feature diagram is a tree-like structure where a feature is represented by a node and sub-features by children nodes. In basic feature models, there are four kinds of relationships between features:

- **Mandatory**: The relationship between a child feature and a parent feature is mandatory when the child is included in all the products in which its parent exists.
- **Optional**: The relationship between a child feature and a parent feature is optional when the child feature may or may not be present in a product when its parent feature is selected.
- **Alternative**: The relationship between a set of child features and a parent feature is alternative when only one feature of the children can be included in a product where its parent is present.
- **Or**: The relationship between a set of child features and a parent feature is an or-relationship when one or more of them can be selected in the products in which its parent feature exists.

Variation Point: Variation points are places in a design or implementation that identify the locations at which variation occurs [20].

Variant: It is a single option of a variation point and is related to the latter using a variability dependency [21].

Specification: Requirements specification is a description of the intended behavior of a software product. It contains the details of all the features that have to be implemented during an evolution of the system.

B. The Framework in a Nutshell

With the large number of features in the SPLs, the manual checking of duplication becomes a complicated and an error-prone task. In order to deal efficiently with the problem of duplication during the evolution of derived products, we proposed the framework depicted in Figure 1 as an attempt to set an automated deduplication tool. As stated in [22], the objective of this framework is to transform the new specifications and the existing feature models into a formal representation, which facilitates the detection of duplication between these two inputs.

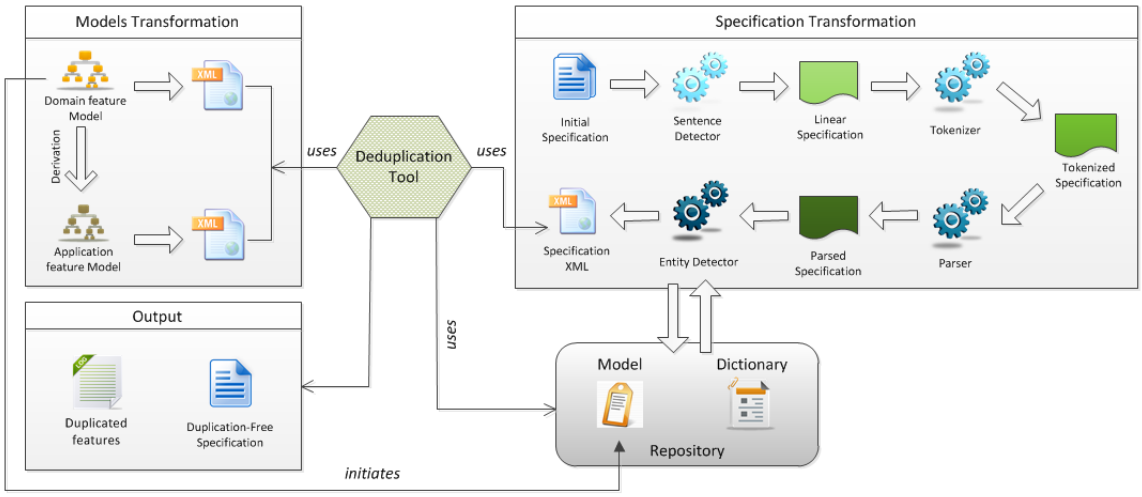

Figure 1. The Overview of the Framework.

1) Models Transformation: Feature-oriented software development (FOSD) [23] is a development paradigm based on the FODA method. This paradigm can be used to develop product lines and to do domain engineering. When other methods lack clean mapping between features, architecture and implementation artefacts, FOSD aims at generating automatically the software products. Hence, tools like GUIDSL [24] and FeatureIDE [25] have been proposed. These tools present features in an interactive form and allow a flexible selection of the features of a derived product.

During this step of the framework, we use the FeatureIDE tool in order to provide a formal presentation of the domain model of the product line and the application model of the derived product. FeatureIDE [25] 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 and generates automatically the corresponding XML source.

2) Specification Transformation: This process of the framework is responsible for transforming the natural language specifications of a derived product to an XML document. For this, we use the OpenNLP library [26], which is a machine learning based toolkit for the processing of natural language text. The remainder of this subsection details the different steps and artefacts involved in this process. In [27], we described the different steps of parsing:

Initial Specification: The main input of this process is the specification of a new evolution related to a derived product. The specification contains the features that have to be implemented in this specific product, expressed in natural language. In the context of our framework, we consider that a feature is the association of a variation point and a variant.

Sentence Detector: The first step of the process consists of detecting the punctuation characters that indicate the end of sentences. After the detection of all sentence boundaries, each sentence is written in a separated line. The output of this operation is a new specification that contains a sentence per line.

Tokenizer: This step consists of dividing the resulted sentences of the previous step into tokens. A token can be a word, a punctuation, a number, etc. At the end 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.

Parser: The parser analyses the sentences of the specification in order to determine the roles of the different tokens based on the rules of the language grammar (e.g., noun, verb, adjective, adverb). In our case, the language used in the 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 syntactic structure of the sentence. Figure 2 illustrates an example of parsing for a natural language requirement related to the case study presented in Figure 5.

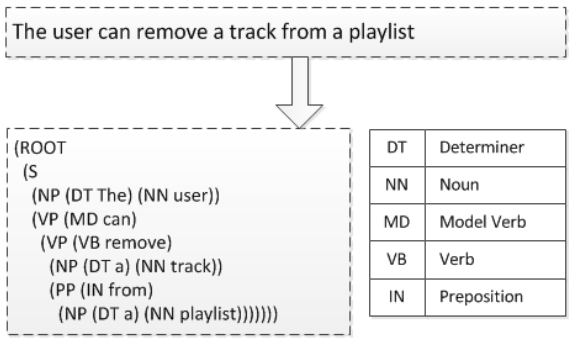

Figure 2. An Example of Syntax Parsing

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

Entity Detector: During this step, we detect semantic

entities in the specification. In our study, we are interested in the parts of the sentences considered as variation points and variants. To carry out this task, we use the repository and especially the model where all the domain specifications are tagged.

The example given in Figure 3 represents a sentence processed by the entity detector. The output is a sentence with markup for the detected variant.

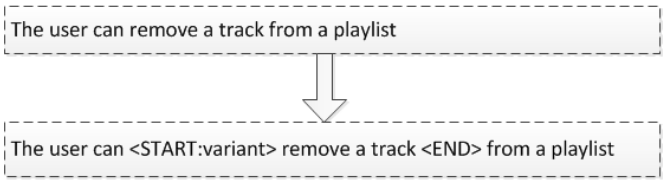

Figure 3. An Example of Detecting Entities

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

3) Repository: The repository contains two components:

- **The model:** It is initially created based on the domain model of the SPL and contains the different features classified by categories, especially <variation point> and <variant>.
- **The dictionary:** It 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 follows the evolution of the product line and its derived products, the new concepts detected in the specification are added to the initial repository.

4) Deduplication Tool: This tool contains a set of algorithms of features verification. In this paper, we focus on the algorithm of deduplication between the feature models and the specifications of new evolutions. The other algorithms are responsible for the verification of non duplication in specifications [27] and in feature models. Before describing the algorithm, we need to define some predicates.

Equivalence: We consider that a variation point (resp. a variant) is equivalent to another variation point (resp. variant) if they both implement the same functionality, which means that they have the same semantics. The equivalents of the different variation points and variants of the system are stored in the repository.

Example: The variant "On-line Sales" associated to the variation point "Sales" is equivalent to the new variant "e-sales".

Duplication: We consider that a feature of the specification is duplicated if the associated variation point and variant have equivalents in the application model or the domain model.

The aim of the algorithm is thus to verify the non-duplication of all the features of the initial specification in order to generate a new correct specification. Indeed, for each feature of the initial specification, the algorithm verifies whether the associated variation point and variant have equivalents in the domain model and the application model. The detection of equivalence is carried out based on the *Repository* content.

5) Output: The output of the framework is a duplication-free specification that contains distinct features, and the list of the duplicate features. The two outputs are sent to the user in order to verify his initial requirements and change them in case of need.

IV. An Algorithm for Duplication-Free SPL

In this section, we provide the formalization of the basic concepts used in the framework, then we describe the deduplication algorithm. The inputs of the algorithm, as depicted in Figure 1, are the XML trees of the domain model, the application model and the specification.

A. Formalizing the Basic Concepts

Prior to explaining the algorithm, a certain number of predicates must be defined. We denote by D the domain model. PD is the set of variation points of D, and VD is the set of variants of D.

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

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

Thus:

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

Similarly, we denote by A the application 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:

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

As a reminder:

$$PA \subseteq PD \quad \text{and} \quad VA \subseteq VD$$

Finally, 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 P and V are not subsets of PA and VA.

B. The Algorithms of Duplication Detection

In order to verify whether a feature is duplicated when implementing a new specification, we propose two algorithms [22]. The first algorithm (Algorithm 1) carries out a comparison between the specification and the domain model. In the second algorithm (Algorithm 2), the comparison is between the specification and the application model of a derived product.

Algorithm 1 Detecting duplication between S and D

```
Principal Lookup :
for each p_i ∈ P do
  for each pd_k ∈ PD do
    if Equiv(p_i, pd_k) then
      Secondary Lookup :
      for each v_ij ∈ V_i do
        for each vd_kl ∈ VD_k do
          if Equiv(v_ij, vd_kl) then
            NoticeDuplication(p_i, v_ij, pd_k, vd_kl)
            Continue Secondary Lookup
          end if
        end for
      end for
      Continue Principal Lookup
    end if
  end for
end for
```

Algorithm 2 Detecting duplication between S and A

```
Principal Lookup :
for each p_i ∈ P do
  for each pa_k ∈ PA do
    if Equiv(p_i, pa_k) then
      Secondary Lookup :
      for each v_ij ∈ V_i do
        for each va_kl ∈ VA_k do
          if Equiv(v_ij, va_kl) then
            NoticeDuplication(p_i, v_ij, pa_k, va_kl)
            Continue Secondary Lookup
          end if
        end for
      end for
      Continue Principal Lookup
    end if
  end for
end for
```

The first loop of these algorithms consists of comparing each variation point of the specification with the variation points of the feature model. When an equivalent is found, the algorithms start another comparison between the variants corresponding to the variation point of the specification and the variants related to the equivalent variation point of the feature model. If a variant is detected, the feature corresponding to the pair (variation point, variant) is considered as duplication.

Although the two algorithms are similar, but it is necessary to implement them both because the decision when detecting a duplication will be different. Indeed, if the new feature exists already in the application model, nothing has to be done. However, if the new feature exists in the domain model but not in the application model, a derivation of this feature is necessary.

Result:

For each algorithm, the duplicate features are stored in a log file that will be sent to the customer in order to inform him of the duplication and its location. When the customer confirms the duplication, the duplicate features are removed from the initial specification and a new duplication-free specification is generated.

V. Automated Tool

At the aim of automatizing the proposed framework, a support tool is currently under development [28]. In this section, we describe the implementation of this tool. Then, we present some of the features it provides.

A. Tool Implementation

The tool is a thick-client java-based application built around Eclipse IDE. A major factor of this choice is the adaptability of Eclipse and its large community of plugins compared to its competitors. Figure 4 presents the architecture of the proposed tool.

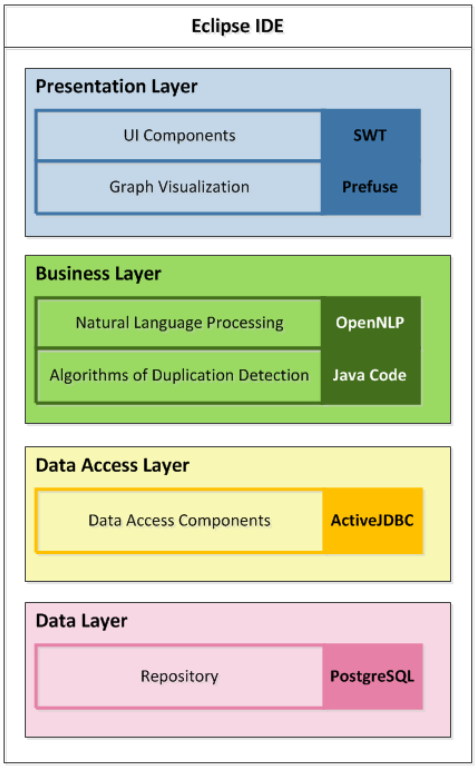

Figure 4. The Tool Architecture.

The tool is composed of four layers:

- **Presentation Layer:** This layer enables the communication between the user and the application. The tool interface is created using SWT [29], which is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. In our case, the application is built on Windows. So, SWT will use Windows facilities to create the interface. Another aspect managed in this layer is the visualization of the processed specifications as the form of a graph. For this, we use Prefuse [30], which

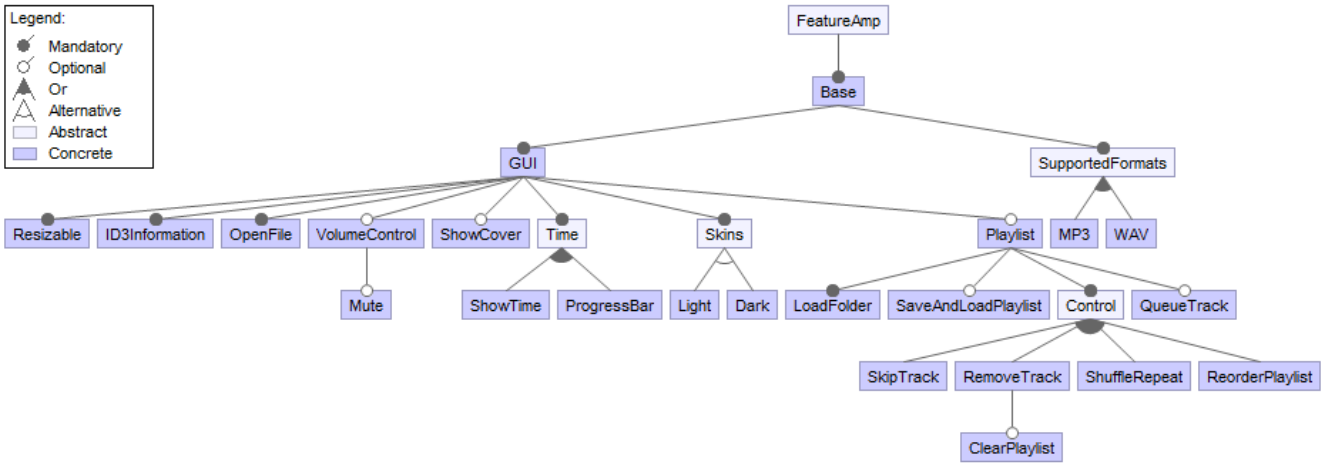

Figure 5. The Domain Feature Model of FeatureAMP.

is an open source toolkit that provides a visualization framework for the Java programming language.

- **Business Layer:** This layer is responsible for the definition of business operations. In our tool, we process textual specifications and we transform them to a tree-like document using OpenNLP. In addition, we implement the algorithms of duplication detection using Java code.
- **Data Access Layer:** This layer provides data to the Business Layer and updates the Data Layer with new information. For this, we use ActiveJDBC [31].
- **Data Layer:** In order to store the content of the repository, we use PostgreSQL [32], which is an open source object-relational database system. In the repository are stored all the domain features, their categories, and their synonyms.

B. Tool Features

The final goal of the automated tool is to detect duplication between new specifications and feature models during SPL evolution. Thus, the main features of the tool are as follows:

- The upload of a textual specification and an XML feature model.
- The transformation of a specification into a tree.
- The detection of duplicate features in a specification.
- The detection of duplicate features in a feature model.
- The comparison of features between a feature model and a specification to detect duplication between them.
- The creation and update of the repository.

The tool provides also some auxiliary features that help achieve the target goal:

- The visualization of the processed specification as a graph.
- The distinction of duplicate features with a different color in the graph.
- The binding of new variants with the corresponding variation points in the repository.
- The manual and automatic update of the repository based on a new specification or a new feature model.
- The re-processing of the specification after a modification of the repository.
- The generation of a log with information about the detected duplicate features.
- The sending of the log to the user via email.

VI. Case Study

In a previous work [28], we evaluated our framework using a CRM SPL. In the feature model of this tool, the features are expressed in the form of sentences. In this paper, we will apply the framework on a different case study which is the FeatureAMP tool [33]. FeatureAMP is an open source audio player product line. The features of this tool are expressed using one word, which makes our task more difficult, because we have to transform the word into a sentence before comparing it to the new requirements.

A. The Feature Models

Figure 5 depicts the domain feature model of FeatureAMP. This tool supports two formats, MP3 and WAV, and provides many functionality such as the play-list management and the volume control.

The main interface of our tool presented in Figure 6 contains two main entries (File) and (Repository). The first entry enables the import of the specification and the feature model. The second entry is used to display the content of the repository.

In order to upload the XML source of the domain model, we use the option ”Open Feature Model”. The interface presented in Figure 7 displays the uploaded feature model and

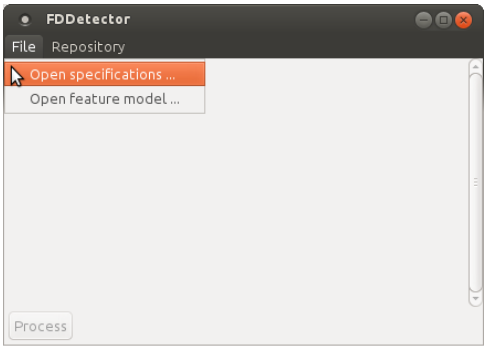

Figure 6. The Main Interface of the Automated Tool.

gives the possibility to load it into the repository. Initially, the creation of the repository was done manually, but its update can be automatic to load the new features from specifications or application feature models.

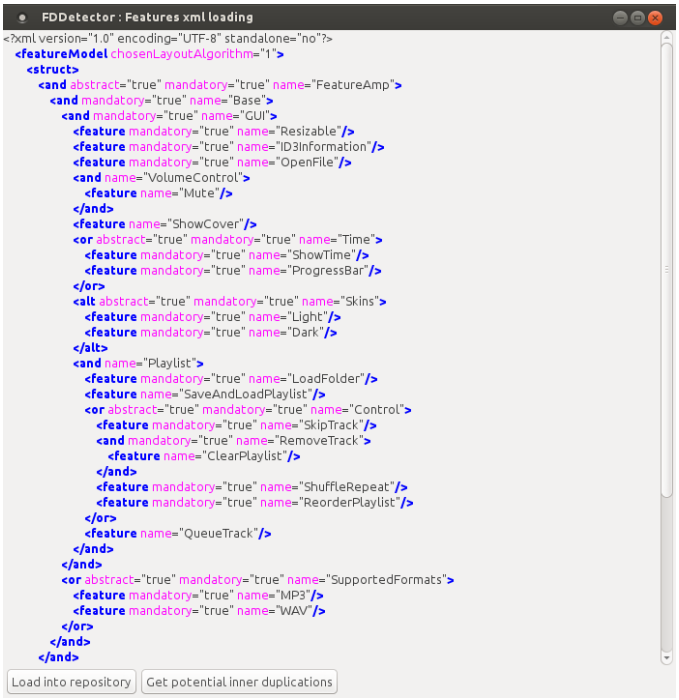

Figure 7. The Feature Model Loading.

B. The Specification

We consider the specification of a new evolution of the product line. The specification is displayed in Figure 8. It contains new features to be introduced in the application, but we added intentionally other duplicate features that already exist in the domain model.

When we press the button ”Process”, the specification is processed and transformed into an XML document. In Figure 9 is presented the graphic form of the generated XML. This presentation facilitates the visualization of the different new features introduced by the specification. In addition, the graph distinguishes the new variants added by the specification by relating them to the node ”unbound variants”. The user will

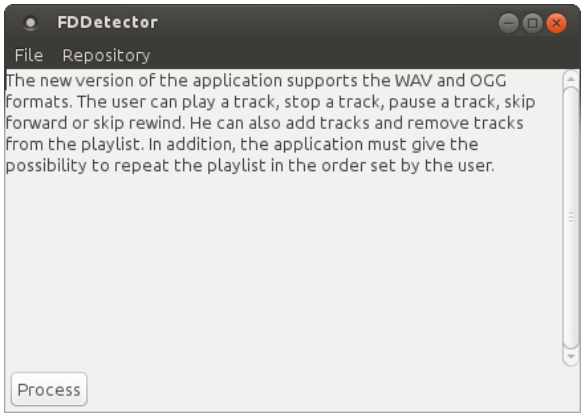

Figure 8. The Specification of the new evolution.

be able to bind these variants with a variation point from the repository.

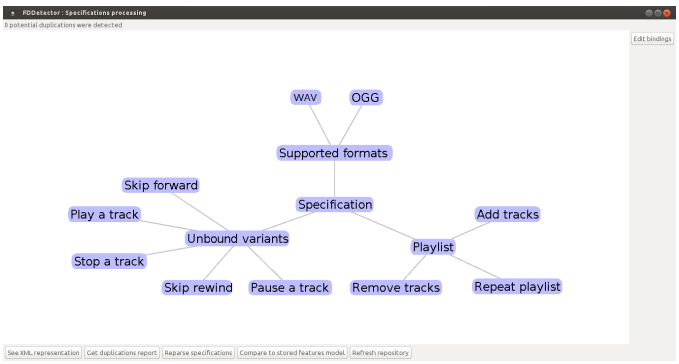

Figure 9. The graph corresponding to the specification.

This interface provides other features such as the display of the generated XML, the re-processing 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.

C. The Duplication Detection

The graph generated for the specification facilitates the distinction of duplicate features inside the specification by presenting them in a different color. For our test, the specification does not contain duplications as displayed in the top left corner of the interface.

If we want to search duplications between the specification and the feature model, we use the button ”Compare to feature models”. The result of this operation is illustrated in Figure 10. Two duplications are detected in the domain model, which represents 20% of the features in the specification. The percentages calculated in this interface will allow us to estimate the gain in the development cost. To visualize the details of the detected duplications, we can generate the log.

In this test, we detected the duplication between the specification and the domain model, but the same test can be performed between the specification and the application

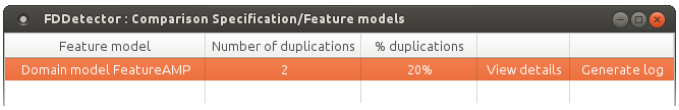

Figure 10. The result of duplication detection.

model. Besides, it has to be noted that in this paper, we focused on the evaluation of the efficacy of the proposed framework, which means, whether the framework allows the detection of duplication or not. Ongoing work consists of testing the effectiveness of the solution by applying it to a complex product line and carrying out a quantitative evaluation to estimate the effort gained by using the proposed approach.

VII. Related Work

In this section, we provide an overview of the studies most relevant to our work by categorizing them according to the issues addressed in this paper.

A. Evolution of Feature and Variability Models

A plethora of studies have dealt with evolution of feature and variability models. For instance, in order to reduce complexity and improve the maintenance of variability in large-scale product lines, Dhungana et al. [34] proposed a method to organize product lines as a set of interrelated model fragments that define the variability of particular parts of the system, and presented a support to semi-automatically merge the different fragments into a complete variability model. The same approach was proposed by Pleuss et al. [35] for feature models.

White et al. [36] presents a new approach for handling feature model drift, which represents the problem of introducing one or more changes in a feature model's constraints. For this, they propose a technique called MUSCLES that consists of transforming multi-step feature configuration problems into Constraint Satisfaction Problems (CSPs), then uses a constraint solver to generate a series of configurations that meet the multi-step constraints.

Cordy et al. [37] defined two particular types of features, regulative features and conservative features, and explained how the addition of these features to the SPL can reduce the overhead of model-checking.

The common denominator of the cited studies is that they all consider evolution in domain engineering, while our approach deals with evolution in application engineering.

B. Model Defects in SPL

Several papers in the literature have addressed model defects caused by SPL Evolution. For example, Guo and Wang [38] proposed 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.

Romero et al. [39] introduced SPLEmma, a generic evolution framework that enables the validation of controlled SPL evolution by following a Model Driven Engineering approach. This study focused on three main challenges: SPL consistency during evolution, the impact on the family of products and SPL heterogeneity.

In [40], 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 equivalence and not only equality of features.

In order to locate inconsistency in the domain feature model of a SPL, Yu [41] 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 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.

Kamalrudin et al. [42] use the automated tracing tool Marama that gives the possibility to 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. Unlike our approach, this one focuses on use cases instead of feature models.

Another approach proposed by Barreiros and Moreira [43] consists of including soft constraints in a feature model, which brings additional semantics that allow improved consistency and sanity checks. Hence, a framework is provided that injects soft constraints into publicly available feature models and recreates typical patterns of use. These features are then subjected to automated analysis to assess the prevalence of the proposed method. While this approach deals with constraint inconsistency, our own one focuses on feature duplication.

C. Evolution in Application Engineering

In the literature, the evolution in domain engineering have been discussed in many papers, while few studies focuses on application engineering. Among these studies, Carbon et al. [44] presented an empirical study, which consists of adapting the planning game to the product line context in order to introduce a lightweight feedback process from application to family engineering at Testo, but it does not provide a general approach that is applicable to all SPLs.

Hallsteinsen et al. [45] introduced the concept of Dynamic Software Product Lines (DSPL), which provides mechanisms

for binding variation points at runtime in order to keep up with fluctuations in user needs. This approach does not explain in details how the variability is managed between application and domain engineering.

Thao [46] proposed a versioning system to support the evolution of product lines and change propagation between core assets and derived products. However, this study also does not provide a method to manage features in application engineering.

A novel approach proposed by [47] analyses the co-evolution of domain and application feature models. It is based on cladistics classification used in biology to construct the evolutionary trees of the different models, then compares the trees using mathematical analysis and provides an algorithm to restore the perfect co-evolution of the software product line and its products.

Our approach is different from the cited approaches because it provides a feature-oriented approach to manage the evolution of derived products in a way that ensures non-duplication in the SPL feature models.

VIII. Conclusion and Future Work

In the literature, many studies have addressed the evolution in SPLs, but the majority of them focused on the domain engineering phase, while application engineering has not been thoroughly discussed. Based on industrial experience, products are also likely to evolve even after their derivation, and this evolution can cause many problems especially duplication in the different artefacts of the product line. In most software engineering projects, evolutions are written in the form of natural language specifications because it is the simplest way for customers to express their requirements. At the aim of avoiding duplication when introducing the new features of these specifications into the existing SPL feature models, we proposed in this paper a framework with two main objectives. The first objective is to transform the feature models and the specifications to a more formal representation, and the second objective is to apply an algorithm that compares the new features proposed in the specifications with the features of the existing models in order to detect feature duplication. At the aim of instantiate the framework, we have started the development of an automated tool whose architecture was described in this paper. The evaluation of this tool was performed using the FeatureAMP product line.

In a future work, we intend to apply the framework on a large scale product line, which will enable us to carry out a quantitative evaluation to prove the effectiveness of our solution.

References

[1] A. Khtira, A. Benlarabi, and B. El Asri, "Towards Duplication-Free Feature Models when Evolving Software Product Lines," Proc. 9th International Conference on Software Engineering Advances (ICSEA'14), Oct. 2014, pp. 107-113.

[2] K. Pohl, G. Böckle, and F. Van Der Linden, Software Product Line Engineering Foundations, Principles, and Techniques, Berlin, Germany: Springer-Verlag, 2005.

[3] N. H. Madhavji, J. Fernandez-Ramil, and D. Perry, Software Evolution and Feedback: Theory and Practice, John Wiley & Sons, 2006, ISBN 978-0-470-87180-5.

[4] Y. Xue, Z. Xing, and S. Jarzabek, "Understanding feature evolution in a family of product variants," Proc. 17th Working Conference on Reverse Engineering (WCRE'10), IEEE, Oct. 2010, pp. 109-118.

[5] S. Urli, M. Blay-Fornarino, P. Collet, and S. Mosser, "Using composite feature models to support agile software product line evolution," Proc. 6th International Workshop on Models and Evolution, ACM, Oct. 2012, pp. 21-26.

[6] A. Ahmad, P. Jamshidi, and C. Pahl, "A Framework for Acquisition and Application of Software Architecture Evolution Knowledge," ACM SIGSOFT Software Engineering Notes, vol. 38, no. 5, Sept. 2013, pp. 65-71.

[7] C. Seidl, F. Heidenreich, and U. Assmann, "Co-evolution of models and feature mapping in Software Product Lines," Proc. SPLC'12, ACM, New York, USA, 2012, Vol. 1, pp. 76-85.

[8] J. Holtmann, J. Meyer, and M. von Detten, "Automatic validation and correction of formalized, textual requirements," In 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11), IEEE, Mar. 2011, pp. 486-495.

[9] A. Fatwanto, "Software requirements specification analysis using natural language processing technique," In 2013 International Conference on QiR (Quality in Research), IEEE, June 2013, pp. 105-110.

[10] M. G. Ilieva and O. Ormandjieva, "Automatic transition of natural language software requirements specification into formal presentation," In Natural Language Processing and Information Systems, Springer Berlin Heidelberg, 2005, pp. 392-397.

[11] P. Clements and L. Northop, Software Product Lines - Practices and Patterns, Boston: Addison-Wesley, 2002.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, Nov. 1990.

[13] C. Salinesi, R. Mazo, O. Djebbi, D. Diaz, and A. Lora-Michiels, "Constraints: the Core of Product Line Engineering," Proc. RCIS'11, IEEE, Guadeloupe-French West Indies, France, May 19-21, 2011, pp. 1-10.

[14] A. Hunt and D. Thomas, The pragmatic programmer: from journeyman to master, Addison-Wesley Professional, 2000.

[15] B. Meyer, "On formalism in specifications," IEEE Software, vol. 2, no. 1, pp. 6-26, 1985.

[16] G. Lami, S. Gnesi, F. Fabbrini, M. Fusani, and G. Trentanni, "An automatic tool for the analysis of natural language requirements," Informe tcnico, CNR Information Science and Technology Institute, Pisa, Italia, Sept. 2004.

[17] IEEE, IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE Computer Society, 1990.

[18] K. C. Kang et al., "FORM: A feature-oriented reuse method with domain-specific reference architectures," Annals of Software Engineering, vol. 5, no. 1, 1998, pp. 143-168.

[19] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, New York, USA: ACM Press/Addison-Wesley, 2000.

[20] I. Jacobson, M. Griss, and P. Jonsson, Software Reuse. Architecture, Process and Organization for Business Success, Addison-Wesley, ISBN: 0-201-92476-5, 1997.

[21] S. Creff, "Une modélisation de la variabilité multidimensionnelle pour une évolution incrémentale des lignes de produits," Doctoral dissertation, University of Rennes 1, 2003.

[22] A. Khtira, A. Benlarabi, and B. El Asri, "Duplication Detection When Evolving Feature Models of Software Product Lines," Information, vol. 6, no. 4, pp. 592-612, Oct. 2015.

[23] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," Journal of Object Technology (JOT), vol. 8, pp. 49-84, 2009.

[24] D. Batory, "Feature Models, Grammars, and Propositional Formulas," Proc. The International Software Product Line Conference (SPLC), Springer-Verlag, vol. 3714 of Lecture Notes in Computer Science, pp. 7-20, 2005.

[25] C. Kastner et al., "FeatureIDE: A Tool Framework for Feature-Oriented

Software Development," Proc. 31st International Conference on Software Engineering, 2009, pp. 611-614.

[26] The Apache Software Foundation, "OpenNLP," opennlp.apache.org.

[27] A. Khtira, A. Benlarabi, and B. El Asri, "Detecting Feature Duplication in Natural Language Specifications when Evolving Software Product Lines," Proc. The 10th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'15), Apr. 2015.

[28] A. Khtira, A. Benlarabi, and B. El Asri, "A Tool Support for Automatic Detection of Duplicate Features during Software Product Lines Evolution," In IJCSI International Journal of Computer Science Issues, vol. 12, no. 4, pp. 1-10, July 2015.

[29] The Eclipse Foundation, "SWT: The Standard Widget Toolkit", eclipse.org/swt/ [retrieved: July, 2015].

[30] The prefuse visualization toolkit, prefuse.org/ [retrieved: July, 2015].

[31] JavaLite, "ActiveJDBC", javalite.io/activejdbc/ [retrieved: August, 2015].

[32] The PostgreSQL Global Development Group, "About PostgreSQL", postgresql.org/about/ [retrieved: July, 2015].

[33] SPL2go, "FeatureAMP", spl2go.cs.ovgu.de/projects/59 [retrieved: August, 2015].

[34] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer, "Structuring the modeling space and supporting evolution in software product line engineering," Journal of Systems and Software, vol. 83, no. 7, 2010, pp. 1108-1122.

[35] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski, "Model-driven support for product line evolution on feature level," Journal of Systems and Software, vol. 85, no. 10, 2012, pp. 2261-2274.

[36] J. White et al., "Evolving feature model configurations in software product lines," Journal of Systems and Software, vol. 87, pp.119-136, 2014.

[37] M. Cordy, A. Classen, P. Y. Schobbens, P. Heymans, and A. Legay, "Managing evolution in software product lines: A model-checking perspective," Proc. 6th International Workshop on Variability Modeling of Software-Intensive Systems, ACM, Jan. 2012, pp. 183-191.

[38] J. Guo and Y. Wang, "Towards consistent evolution of feature models," In. Software Product Lines: Going Beyond, Springer Berlin Heidelberg, 2010, pp. 451-455.

[39] D. Romero et al., "SPLEMMA: a generic framework for controlled-evolution of software product lines," Proc. 17th International Software Product Line Conference co-located workshops, ACM, 2013, pp. 59-66.

[40] R. Mazo, "A generic approach for automated verification of product line models," Ph.D. thesis, Pantheon-Sorbonne University, 2011.

[41] D. Yu, P. Geng, and W. Wu, "Constructing traceability between features and requirements for software product line engineering," In 19th Asia-Pacific Software Engineering Conference (APSEC'12), IEEE, vol. 2, pp. 27-34, Dec. 2012.

[42] M. Kamalrudin, J. Grundy, and J. Hosking, "Managing consistency between textual requirements, abstract interactions and Essential Use Cases," In Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual, IEEE, July 2010, pp. 327-336.

[43] J. Barreiros and A. Moreira, "Soft Constraints in Feature Models: An Experimental Assessment," International Journal On Advances in Software, vol. 5, no. 3 and 4, 2012, pp. 252-262.

[44] R. Carbon, J. Knodel, D. Muthig, and G. Meier, "Providing feedback from application to family engineering-the product line planning game at the testo ag," Proc. 12th International Software Product Line Conference (SPLC'08), IEEE, Sept. 2008, pp. 180-189.

[45] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," Computer, vol. 41, no. 4, 2008, pp. 93-95.

[46] C. Thao, "Managing evolution of software product line," Proc. 34th International Conference on Software Engineering (ICSE'12), IEEE, Jun. 2012, pp. 1619-1621.

[47] A. Benlarabi, A. Khtira, and B. El Asri, "An Analysis of Domain and Application Engineering Co-evolution for Software Product Lines based on Cladistics: A Case Study," Proc. 9th International Conference on Software Engineering Advances (ICSEA'14), Oct. 2014, pp. 495-501.