

# Feature Patterns and Multi-Paradigm Variability Models

(GIRO Technical Report 2008/01-v0.91)

Miguel A. Laguna, Bruno González-Baixauli, José M. Marqués, Rubén Fernández

Department of Computer Science, University of Valladolid,  
Campus M. Delibes, 47011 Valladolid, Spain  
{mlaguna, bbaixauli, jmmc, luifern}@infor.uva.es

**Abstract:** One of the most important issues in the development of software product lines is the elicitation, management, and representation of the variability. In this context, feature models are the basic instrument to analyze and configure the variability and communality of the product line. But a feature model can be considered as an amalgamation of various different variability models (structural, behavior, non functional, or platform variability aspects are combined in a single model). The separation of these different facets can help in the development of the product line. Features, as core model, can be completed with other techniques (i.e. goals or some UML models) for expressing diverse aspects of the variability. The second part of the article explores the possibilities of identifying patterns in the feature models and relates these patterns with the correspondent architectural counterparts. If we define a feature patterns catalog, the automated creation of traceability links between the product line models is possible and hence the productivity in the development process of the product line will be enhanced. This approach allows proceeding in several stages, using the appropriate paradigms (goals, features, package models, platforms...) in each phase of the process. The global picture is a sequence of model transformations from goal/requirements to features and from both to the architecture (a set of UML models). The conclusion is positive as the combination of paradigms makes more straightforward the development process of the product line.

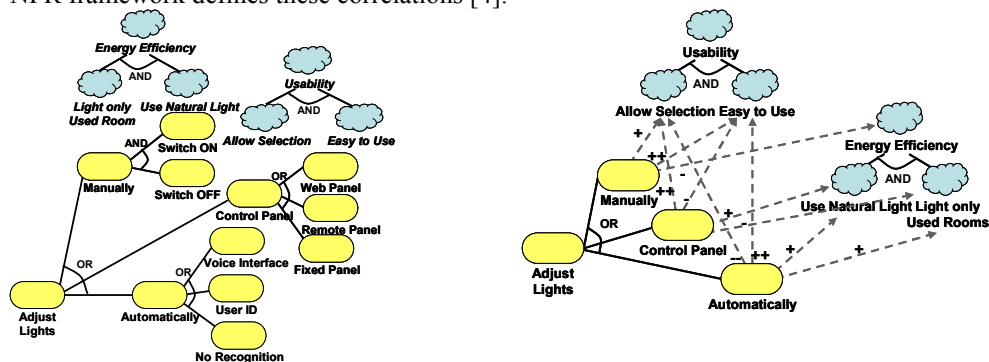
## 1 Introduction

The development of software product lines (PL) faces many technical and organizational trends, in spite of its success in the reuse field [2, 6]. The development of a product line involves two main categories of software artifacts: the artifacts shared by the members of the product line and the product-specific artifacts. The product line itself is a set of reusable assets, where three abstraction levels can be clearly identified (requirements, design and implementation assets). In the requirement level, one of the key activities is the specification of the variability and

communality of the product line. The design of a solution for these requirements constitutes the domain architecture of the product line. Later, in the application engineering process, the application architecture must be derived from the domain architecture. In this process the customer functional and non-functional requirements for a particular application are used for choosing among alternative features. This activity is essentially a transformation process where a set of decisions at the requirements level generates the initial feature product model and, consequently, via traceability paths, the architecture of the product [2].

Therefore, one of the most critical points is the elicitation and analysis of variability in the product line requirements. In addition to the information that expresses the requirements themselves, it is important to know the variability of the requirements, and the dependencies between them. In this context, feature models are the basic instrument to analyze and configure the variability and communality of the software family. But although its effectiveness has been proven in many projects, these models are oriented to the solution more than to requirements. On the other hand, not only the functionality but also non-functional or platform specific aspects must be taken into account. Consequently, the use of the feature diagrams as a monolithic tool over-simplifies and limits the potential of the technique. We propose to use additionally techniques of Goal Oriented Requirements Engineering and Model Driven Engineering (MDE). This proposal assumes that more than a unique view is needed to express the diverse variability aspects of a product line.

The Goal Oriented Requirements Engineering proposes an explicit modeling of the intentionality of the system (the “whys”). Intentionality has been widely recognized as an important point of the system, but it is not usually modeled. The main advantages of the goal-oriented approach are that it can be used to study alternatives in software requirements (it uses AND/OR models for the different alternatives) and that it can easily relate functional and non-functional requirements (NFR). A goal is an objective that the system under consideration should achieve [32]. There are two types of goals: (hard) goals and *soft-goals*: goal satisfaction can be established through verification techniques, but *soft-goal* satisfaction cannot be established in a clear-cut sense (it is usually used to model non-functional characteristics of the system) [32]. The dependence between goals and *soft-goals* can be established. The NFR framework defines these correlations [4].



**Fig. 1** Variability in goals and soft-goals: Goals (ellipses) and soft-goal (clouds) model for a PDA writing methods and correlation between them

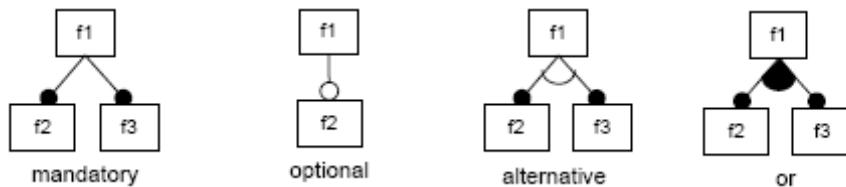
Model Driven Engineering (MDE) is a recent paradigm that bases the software development in models and their transformation to code. The best known approach is Model Driven Architecture (MDA). MDA was introduced by the Object Management Group (OMG) and is based on the Platform Independent Model (PIM) concept, a specification of a system with independence of platforms (e.g. .NET, or J2EE) [27]. The PIM must be transformed into a Platform Specific Model (PSM) [27]. As the main strength of MDE paradigm is the manipulation of models, it is very convenient use its techniques to define the transformations between feature, goal, and UML models of a product line. The aim is to develop the product line (and in parallel each particular product) in several stages, using the appropriate model (goals, features, platforms...) in each phase of the process. The global picture is a sequence of model transformations from goal/requirements to features and from both to the architecture (a set UML models). This approach requires the establishment of precise rules, using meta-modeling and transformations techniques. To establish these rules, the recurring patterns in the goal and feature models must be discovered. All the previous work done in these fields must serve to improve the productivity in the product line development process.

The rest of the paper is as follows: The next discusses the separation of the variability model in several views, enabling to work in several abstraction levels (goals, features, platform independent, and platform specific models). Section 3 identifies the basic patterns that can be found in feature models. The result is a catalog or feature patterns and their corresponding architectural and behavioral UML models. Section 4 shows the utility of the MDE approach on automate these transformations. Section 5 describes the feature modeling tool we have developed in order to automate these transformations. Section 6 presents related work and Section 7 concludes the paper and proposes additional work.

## **2 Multi-paradigm Product Line Requirements**

Originally the Feature Oriented Domain Analysis (FODA) [20] proposed features as the basis for analyzing and representing commonality and variability of applications in a domain. A feature represents a system characteristic realized by a software component. There are four types of features in feature modeling: Mandatory, Optional, Alternative, and Or (Figure 2). A Mandatory feature must be included in every member of a product line family as long as its parent feature is included; an Optional feature may be included if its parent is included; exactly one feature from a set of Alternative features must be included if a parent of the set is included; any non-empty subset of an Or feature set can be included if a parent feature is included. From this initial version, several improvements have been proposed (see [29, 30] for a comparison). In particular a (min:max) cardinality can be used for both ends of the parent/child relations: Mandatory (1:1) and Optional (0:1) are associated with the child part of the relation; Alternative (1:1) and Or (1:n) are associated with the parent part of the relation (referring to a group of n sub-features). As a natural extension, the general cardinality (m, n) can indicate mixed type of feature decompositions. Other extensions [1] add an atomic kind of feature (some of the leaves of the feature tree)

with attribute value (String, Integer, etc.). This aspect is interesting if we seek to transform feature diagrams into architectural models. Schobbens *et al.* [30] have established a formal semantic that covers all the variants of feature diagrams, clarifying most of the ambiguities of the different versions.



**Fig. 2** Basic FODA constructions

But actually the original idea of feature models tries to cover simultaneously different variability models: it represents structural or domain variability, behavior variability, non functional variability, platform variability, etc. The differentiation of these different aspects, keeping them in separate models as complementary views, can improve the development of the product line. Other techniques apart from feature diagrams are better at expressing different aspects of variability, while the feature model acts better as the central piece of the puzzle that connect the rest of the models. It is as well the best tool for the latter configuration process.

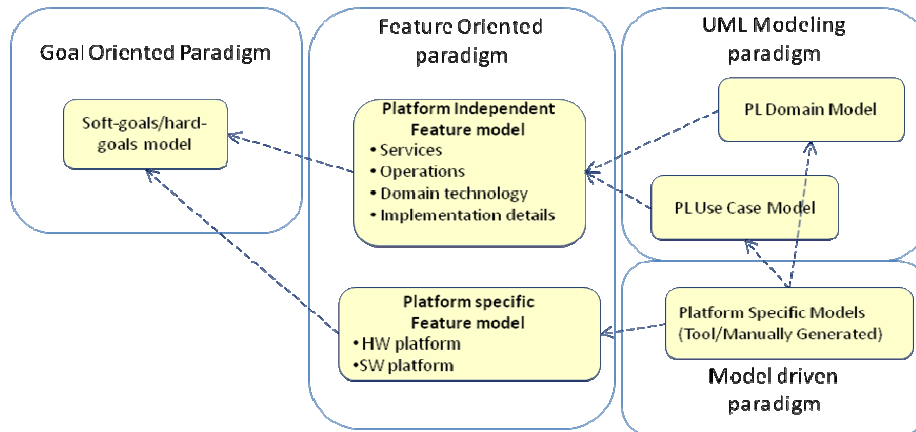
FODA [20] and FORM [19] classify the commonality and variability aspects in:

- The capabilities of applications in a domain from the perspective of the user. They are user visible characteristics that can be identified as services, operations, or non-functional characteristics.
- The operating environments (hardware and software platforms, including operating systems) in which applications are used and operated.
- The application domain technology based on which requirements decisions are made (including laws, standardization, business rules).
- The implementation techniques (algorithms or data structures).

Jarzabek [17] reorganizes the product line requirements in features and quality attributes (NFR). The former can be categorized into behavioral requirements and design decisions:

- Behavioral requirements represent functionality or services. They describe how the system should react to particular inputs and how the system should behave in particular situations.
- Design decision can be categorized into domain technology, implementation techniques and operating environment, in a way similar to FODA.

These different types and sub-types of features can be studied separately. The proposal is to use the best available technique for each aspect, in order to optimize the elicitation and representation of the variability. Figure 3 shows how the combination of techniques and paradigms can serve to this intention.



**Fig. 3** Combination of paradigms in variability analysis

The FODA capabilities category of features includes very different aspects: the structural aspect that can be represented by classical domain models (class diagrams); the behavior facets that can be expressed by use case models; or the non-functional characteristics that can be analyzed better with soft-goals models. *Soft-goals* are specifically a technique to introduce the non-functional aspects in the elicitation and analysis of the requirements. The high level features represent in many occasions the aims of the product line and can be represent better as (hard) goals models, conceived specifically as a way of introducing intentionality. Consequently, these goals and soft-goals will allow introduce a rationale basis in the selection of variants during the product derivation process. Of course, a set of traceability links between *soft-goals*, goals, features, and UML models must be carefully established. A tool that can evaluate these goals and *soft-goals* models automatically with respect to the customer preferences has been built to support this approach [13]. In short, we propose to limit the use of feature models to express structural and functionally variability.

The main function of the feature model is to connect the rest of the techniques and to allow the derivation from more abstract to more refined models. The features relate all the views of the product line requirements: pure requirement models (goal, use cases, constraints and business rules) but also specification models (domain and architectural design models). An obvious consequence is that the feature category is valuable information that can be added to the feature diagrams. In fact, nothing prevents us of assigning several categories to the same feature, associating it with behavioral o structural models (think in a feature group for payment types: the feature group can imply simultaneously a specialization class structure and a use case diagram with extend relationships).

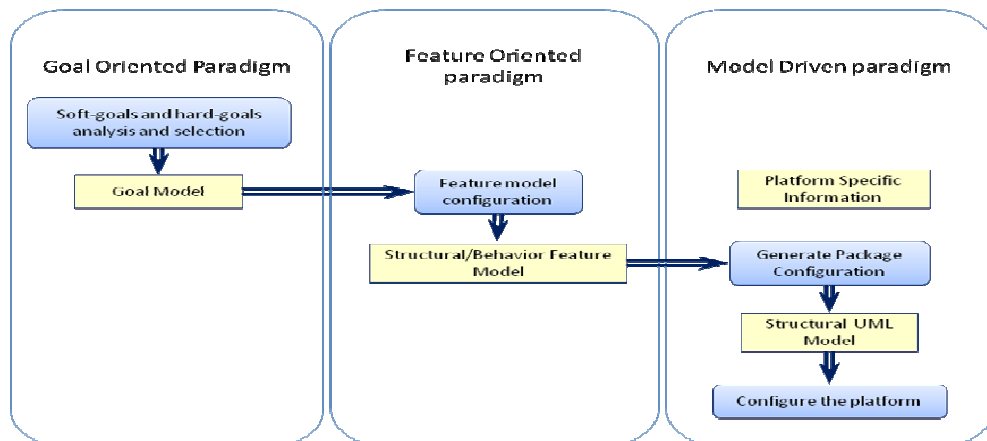
The UML models are conventional diagrams that are organized in packages. Apart from the base package, each optional feature must have a counterpart in a package which includes the set of class diagrams, use cases and sequence diagrams that are the solution that achieve this feature. The packages are related using the UML package merge mechanism. In [22], we explained the application of this technique to the organization and configuration of the product line architecture.

The platform variants must be considered in a second stage, as most of the variation points are independent of the operating environment. In the MDE/MDA paradigm, the main contribution is the separation of the PIM (platform independent model) from the PSM (platform specific model). This approach implies that the operating environment category of features must be analyzed in second term, after the capabilities features have been considered.

Finally the last two groups of features (application domain and implementation techniques) are too specific to introduce significant differences in the general variability analysis. Algorithm implementation details or legal constraints are important information about the common aspects of the product line but not in general from the point of view of the variability analysis.

Therefore, the image of the Figure 1 emerges, as combination of several paradigms:

- The goal (hard goals essentially) models represent the intention of the product line, i.e. the high level objectives the application must solve. The soft-goals of goal models represent the non-functional characteristics. These can be used with the hard-goals contribution information to configure the optimal solution for a concrete application in the product line.
- The feature model represents the end-user functional requirements, connected with the hard goals of the goal models.
- The UML models organize the architecture specification of the product lines, connected with the feature model.
- The features that configure the operating environment must be considered in a later stage, as we adopt the MDE paradigm of separation of the PIM/PSM models.
- The information about the details of frameworks, platforms, etc. is kept apart from the platform independent models.



**Fig. 4** Combination of paradigms in the application derivation process of a product line

Once the product line is developed, the process of product derivation can be described. The goals selected in the high level configuration process determine the

features configuration. This configuration leads to the UML models that define the initial product architecture (class diagrams that represent the concrete domain model of the configured application and a set of use cases that represents the behavior of the application from the user viewpoint). The Figure 2 shows the schematic view of the process of configuration of an application: First, using the tool described in [13] we find the optimal combination of goals and *soft-goals* for the satisfaction of the customer needs. This combination originates the configuration of the feature application model and this last yield the package configuration for the concrete application with the basic architecture. These steps can be totally automated. From here, the two alternative ways are open: manually complete the application or use a MDA code generation tool. The experiences so far consist of manually adding the user interface and persistence details to the UML package models. The platform specific models are based in Microsoft .NET as this platform allows implementing directly the concept of package merge using C# partial classes. This manual approach has been successfully applied to the development of several product lines in the Web and mobile applications domains. An alternative under study is to use code generation tools as AndroMDA or Bridge Point. In this approach, the selected packages are first merged (using MDE transformation tools) and the resulting product model is processed by the generation tool.

But the productivity (and the complexity in non trivial product lines) demands to automate the construction and configuration of the diverse product line models, as the MDE paradigm advocates. The transformation from goal to feature model has been treated by Yu *et al.* in their work on Goal-Oriented requirement engineering [35, 36]. Basically they use a catalog of goal patterns and their corresponding feature constructions. The next section deals with the analogous transformation of the feature models (basically the functional and structural category of features) into architecture level models, including the package organization and a first cut of the package contents.

### 3 Feature Patterns

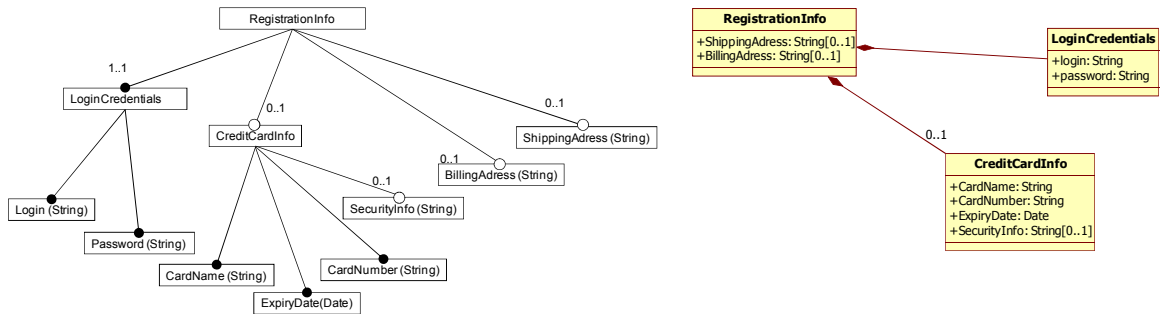
Once the feature diagram of a product line is established, several design level UML models must be developed. Our intention is to build a catalog of commonly used derivations of feature to UML models. A revision of the literature has revealed that it is naive to pretend a simple and univocal transformation from feature models to UML diagrams. Therefore, we have adopted a pragmatic and multi-view approach: separate the different categories of features in a variability model and treat each of these categories in a different way. Sochos *et al.* [31] have reviewed recently the approaches apart from proposing a new one. The classical works of Kang and Lee [19, 20, 23], Czarnecky [8], Griss *et al.* [14], or Bosh [2], between others have allowed to identify a set of feature patterns that potentially can populate the catalog.

We use in this and the next section a selection of examples extracted from a large case study described in [23], from different points of view, using feature models, class models and activity diagrams. In particular, it is an electronic commerce product line and includes a great number of optional features and, in consequence, a large number

of possible derived products. The feature tool used is the *fmp* eclipse plug-in developed in the Waterloo University [1] and an adapted version of a conventional UML CASE tool.

We differentiate two kinds of transformations: the structural information is mapped to class diagrams and the behavioral features are connected with use case diagrams. We can annotate the features as structural or behavioral oriented. Many features can represent both facets and consequently the annotations are not exclusive.

Consider the simplest situation. A feature AND construction that represents structural information can be directly transformed into classes and attributes (Figure 5). In the Czarnecki meta-model approach we base our models [10], the features are typed. The type can be a simple type or a FEATURE type itself. The key aspect is that the leaves of the tree can be features with information about simple types (Czarnecki uses String, Integer, etc.; we use some more elemental types as Date, Time, Money, directly translatable to conventional programming languages) and in these cases a simple typed feature is mapped into an attribute of a parent class.



**Fig. 5** Structural feature model fragment and the corresponding mapped design equivalent

The situation where the leaf is not an atomic feature (or simply is an intermediate node in the tree) requires a little more complicate solution. The general mapping is to create a class that represents the feature (more details will be added during the posterior refinement of the domain model). The mandatory features imply a 1..1 composition relationship and the optional features imply a 0..1 composition relationship. The Figure 5 includes some examples of both circumstances: CreditCard is a class that represents optional information in the class model; LoginCredentials represents compulsory information of the sale.

This supposes the identification of several simple patterns in feature models:

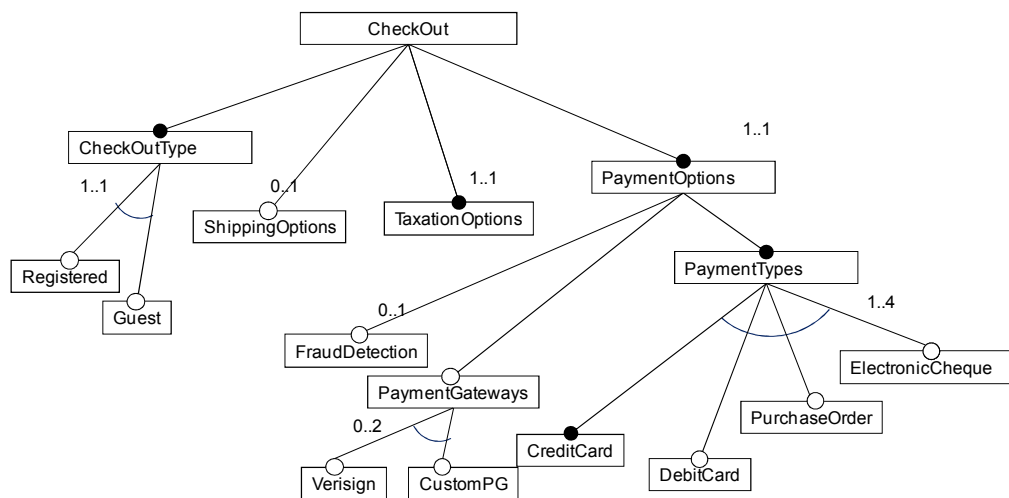
- The presence of a mandatory feature of default FEATURE type originates a class that is associated (with a 1..1 multiplicity) with another class that represents the parent feature.
- The presence of an optional feature of default FEATURE type originates a class that is associated (with a 0..1 multiplicity) with another class that represents the parent feature.



- The presence of a mandatory feature of a simple type, (INTEGER, STRING, DATE..., i.e. any type different of default FEATURE type) originates an attribute in a class that represents the parent feature.
- The presence of an optional feature defined by a simple type originates an optional attribute (represented by the UML attribute multiplicity information) in a class that represents the parent feature.

Summing up, the structure of classes/attributes connected by a 1..1 or 0..1 multiplicity association or composition relationships reflects the original structure of features. Examples of these patterns can be observed in Figure 5.

The most common architectural equivalence of the grouped features (alternative and OR groups) is based in inheritance. Really, a combination of generalization and composition relationships is needed to differentiate alternative from OR patterns. Both variants have initially a common treatment in the correspondent class diagram structure: A composition relationship indicates if the selection of one or several sub-features is compulsory (minimum multiplicity equal or greater than one) or optional (minimum multiplicity equal to zero). The composition maximum multiplicity differentiates the OR relation (maximum multiplicity greater than one) from the pure alternative relation (maximum multiplicity exactly equal to one).



**Fig. 6** Structural feature model fragment with examples of alternative and OR feature constructions

The Figure 6 shows an example of both variants. The *CheckOut* feature has always a *CheckOutType*, but this must be (in a concrete application) a *Registered CheckOut* or, alternatively, a *Guest CheckOut*. The two variants can't be implemented simultaneously in a concrete application. This is reflected in the composition relationship shown in Figure 7: only one instance of type *CheckOutType* can be

connected to a *CheckOut* instance. This is a polymorphic composition, frequent in object oriented models. However the variant of the *PaymentTypes* feature implies that more than one feature can be selected (but one of this must be always the *CreditCard* payment type). The multiplicity 1..4 of the composition relationship states this possibility. Finally, the *PaymentGateways* group of features are optional but compatible (the multiplicity 0..2 clarify the possibilities).

The Table 1 presents the standard conversion patterns of feature to structural models. The table conforms to the widespread solution designs that can be found in the literature. However, we can appreciate that some basic information is missing in this simple approach. For example, it is necessary to show that in some applications the payment type *PurchaseOrder* or *ElectronicCheque* can be used or not, but any application must have the possibility of *CreditCard* *PaymentType*. The problem is that the image of figure 7 and the table 1 try to be a compact representation of the product line, not a UML representation of the set of applications in the product line. This representation must be modified to clarify these differences. Most authors use stereotypes, annotating some classes as variants or optional elements [12, 15, 18] and others use specialized superimposition UML diagrams [7] (see the related work section for details). For example, to solve the mentioned problem about payment types, the *PurchaseOrder* or *ElectronicCheque* classes could have the stereotype <<variant>> to differentiate from the *CreditCard* *PaymentType*, present in all the applications of the product line.

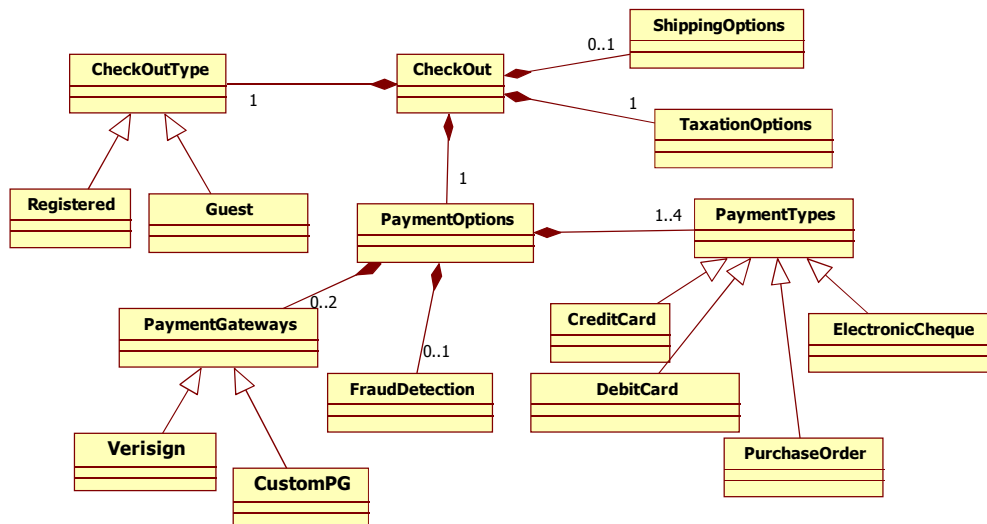
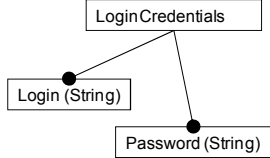
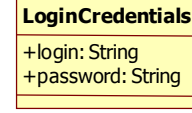
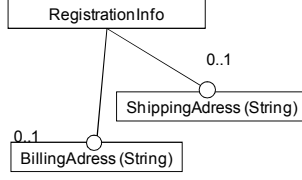
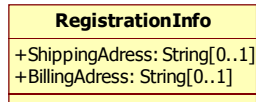
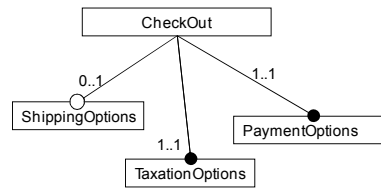
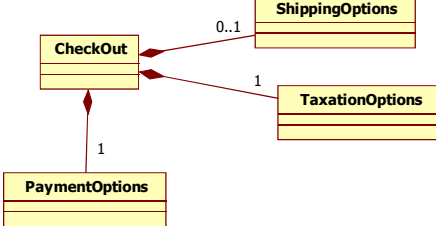
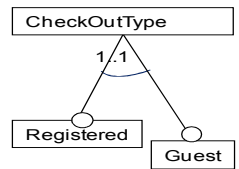
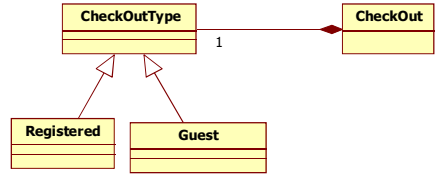
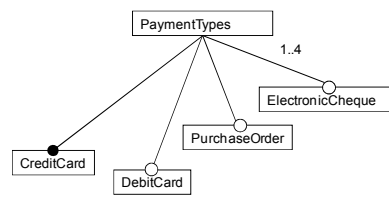
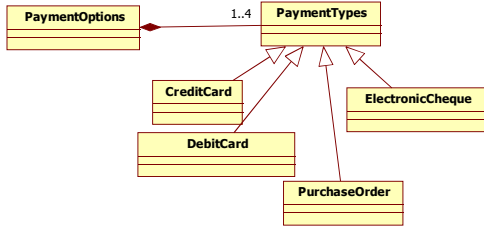
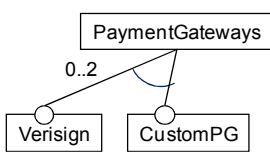
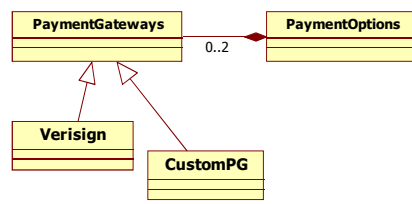


Fig. 7 Product line Structural Feature model fragment and the correspondent package based architectural solution

**Table 1** Some of the basic structural features construction and their mapping into compact class diagrams

Feature Construction	Class Structure (classical version)
 <p>UML Feature Construction diagram for LoginCredentials. The root feature is LoginCredentials, which has two child features: Login (String) and Password (String).</p>	 <p>UML Class Structure diagram for LoginCredentials. The class LoginCredentials has two public attributes: login: String and password: String.</p>
 <p>UML Feature Construction diagram for RegistrationInfo. The root feature is RegistrationInfo, which has two child features: ShippingAddress (String) and BillingAddress (String). Both child features have a cardinality of 0..1.</p>	 <p>UML Class Structure diagram for RegistrationInfo. The class RegistrationInfo has two public attributes: ShippingAddress: String[0..1] and BillingAddress: String[0..1].</p>
 <p>UML Feature Construction diagram for CheckOut. The root feature is CheckOut, which has three child features: ShippingOptions (0..1), TaxationOptions (1..1), and PaymentOptions (1..1).</p>	 <p>UML Class Structure diagram for CheckOut. The class CheckOut has three associations: ShippingOptions (0..1), TaxationOptions (1), and PaymentOptions (1).</p>
 <p>UML Feature Construction diagram for CheckOutType. The root feature is CheckOutType, which has two child features: Registered and Guest. Both child features have a cardinality of 1..1.</p>	 <p>UML Class Structure diagram for CheckOutType. The class CheckOutType has two subclasses: Registered and Guest. There is also an association from CheckOutType to CheckOut with a cardinality of 1.</p>
 <p>UML Feature Construction diagram for PaymentTypes. The root feature is PaymentTypes, which has four child features: CreditCard, DebitCard, PurchaseOrder, and ElectronicCheque. CreditCard has a cardinality of 1, while the others have 1..4.</p>	 <p>UML Class Structure diagram for PaymentTypes. The class PaymentTypes has four subclasses: CreditCard, DebitCard, PurchaseOrder, and ElectronicCheque. There is also an association from PaymentTypes to PaymentOptions with a cardinality of 1..4.</p>
 <p>UML Feature Construction diagram for PaymentGateways. The root feature is PaymentGateways, which has two child features: Verisign and CustomPG. Both child features have a cardinality of 0..2.</p>	 <p>UML Class Structure diagram for PaymentGateways. The class PaymentGateways has two subclasses: Verisign and CustomPG. There is also an association from PaymentGateways to PaymentOptions with a cardinality of 0..2.</p>

We have discussed in our previous work the disadvantages of these approaches and proposed to separate completely the representation of variability of the product line from the residual variability of the concrete applications [22]. We use for this the UML package merge mechanism. The package merge mechanism basically consists of adding details to the models in an incremental way. According to the specification of UML 2, <<merge>> is defined as a relationship between two packages that indicates that the contents of both are combined. It is very similar to the generalization and is used when elements in different packages have the same name and represent the same concept, beginning with a common base. The concept is extended incrementally in each separate package. Selecting the desired packages, it is possible to obtain a tailored definition from all the possible ones. Though the examples in UML focus on class diagrams, the mechanism can be extended to any UML model, in particular use cases and sequence diagrams.

This mechanism allows a clear traceability between feature and UML models to be established. Each optional feature is described by an optional package of the product line that will be selected or not in function of the concrete configuration of features. The process consists of establishing for each UML model, a base package that embodies the common part of the product line. Then, associated to each optional feature, a new package is built, so that all the necessary changes in the model remain located. This package is connected through the <<merge>> relationship with its base package in the exact point of the package hierarchy. The sense of the relationship expresses the dependence between packages: the base or merged package can always be included in a specific product, the receiving package is an extension of the base package and can only be included if the base package is also selected. This is exactly the way the expert decides which features are included or not during the configuration process, and must be directly reflected in the configuration of packages.

A modified version of the Table 1 patterns are imaginable, combining the classical with the package merge based interpretations. The original version uses only the multiplicity of the attributes in classes to represent optional features and is much more compact. But the second version is preferable as removes any ambiguity and is directly mapped to code. Consequently the representation of the product line structural model that we use is the one shown in the Figure 8, instead of the one shown in the previous Figures 5 and 7. The Table 2 presents the package version of the conversion of feature patterns to structural models.

The apparent complexity of the package model reflects the real complexity of the product line itself and it is easily handled by the current CASE and IDE tools. Being realistic, the product lines were we have applied the approach deal with at most dozens of optional features, not hundreds or them. In any case, the automation of the process is absolutely necessary in order to successfully adopt the product line development paradigm. We have used so far the xmp plug-in [1] as configuration tool, combining the output of the plug-in (as a XML file) with the Visual Studio tools and C# compiler and the results are very satisfying. The C# language is selected because the direct support of the merge mechanism by means of partial classes. In [22], these experiences are described and the basic transformations from feature models to packages diagrams are defined using QVT [28].

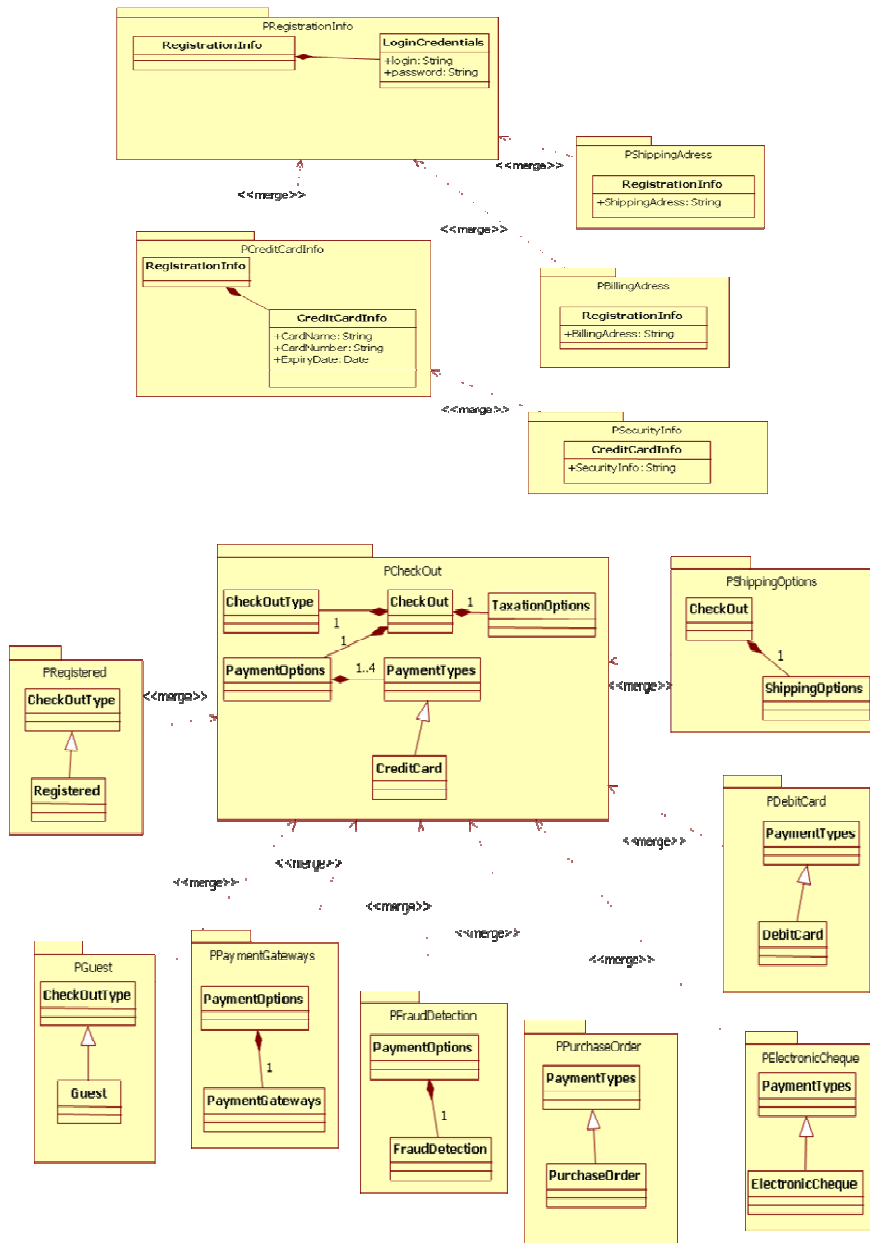
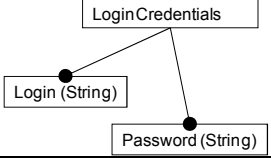
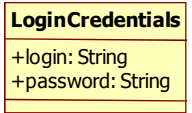
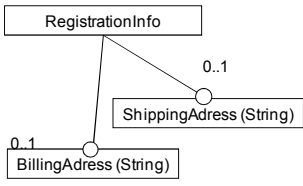
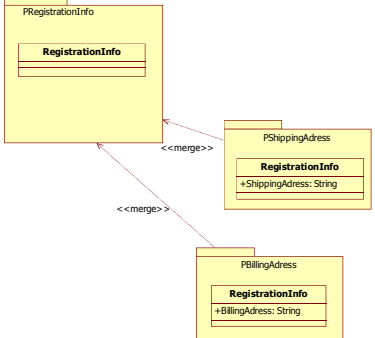
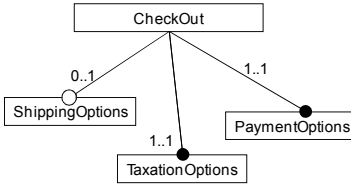
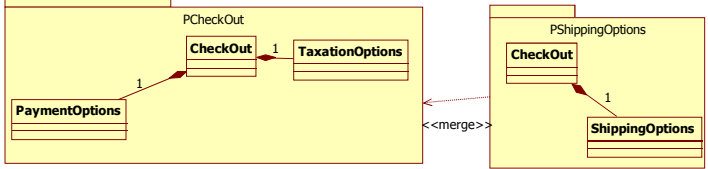
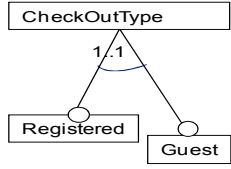
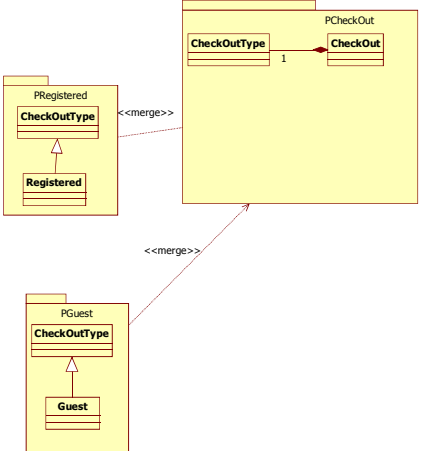
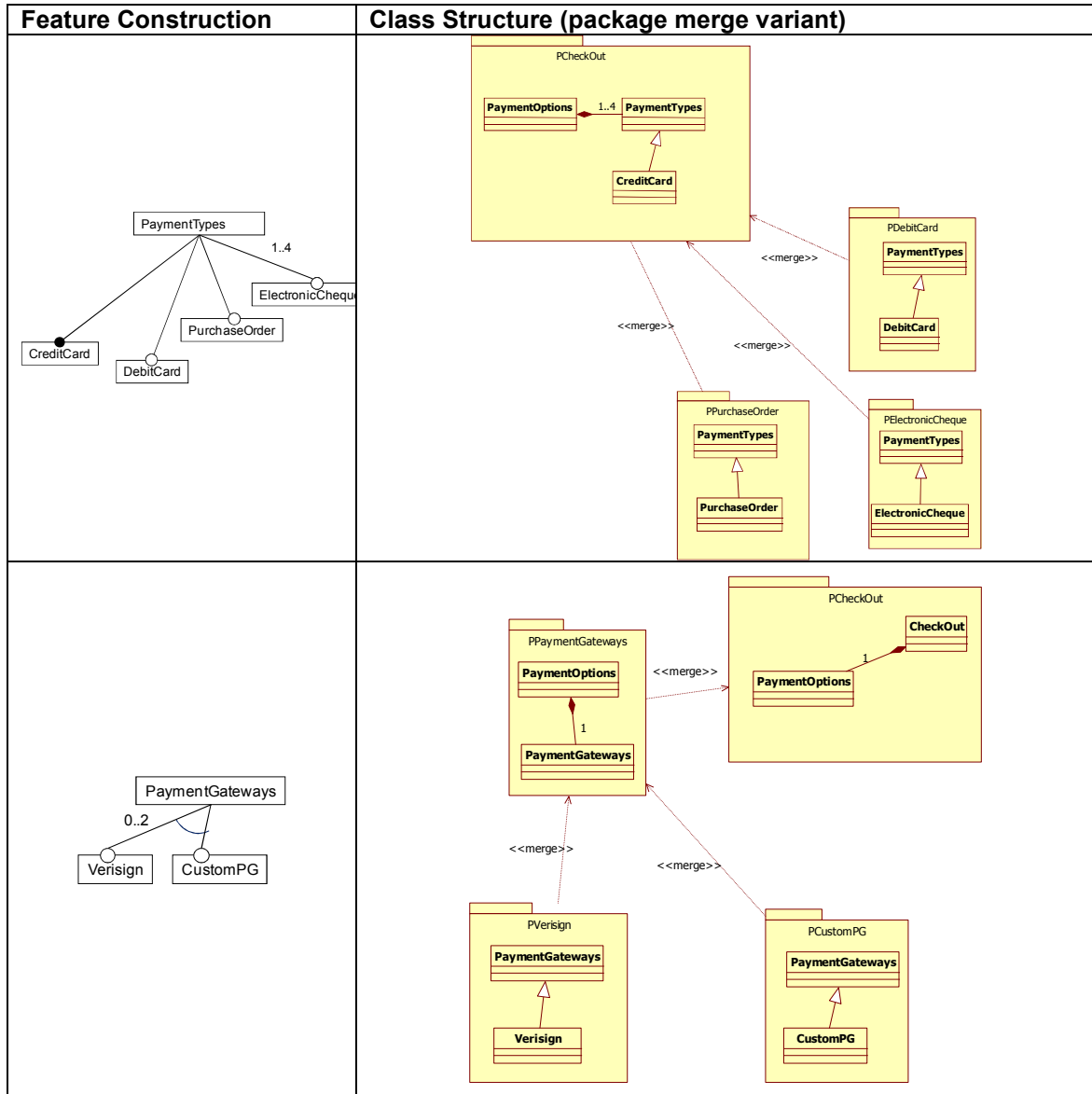


Fig. 8 Product line package based architectural solution (corresponding to the feature model of Figures 5 and 7)

**Table 2** The basic structural features and their translation to package combination of class diagrams

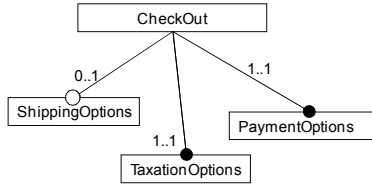
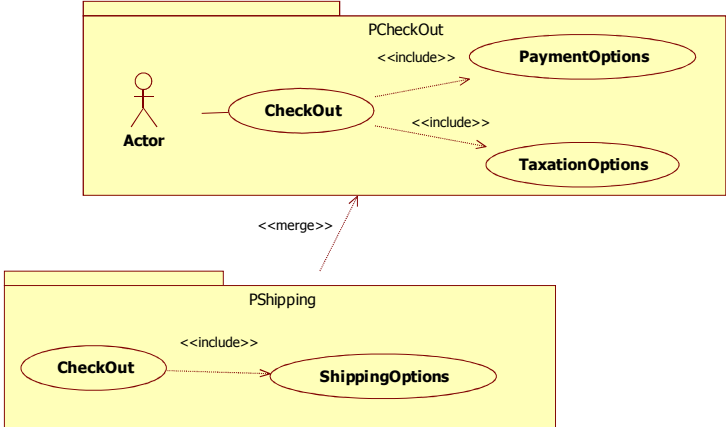
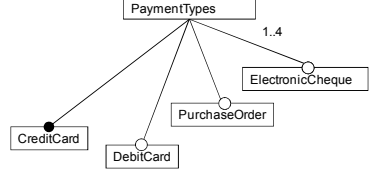
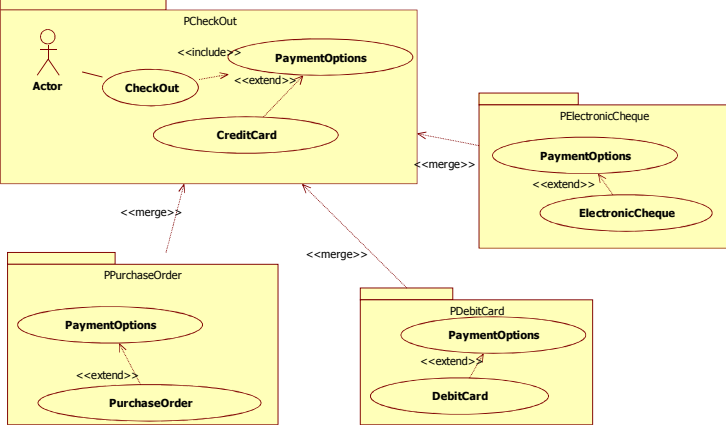
Feature Construction	Class Structure (package merge variant)
 <p>A feature construction diagram for <b>LoginCredentials</b>. It consists of a root node <b>LoginCredentials</b> with two child nodes: <b>Login (String)</b> and <b>Password (String)</b>. Both child nodes are connected to the root by solid lines with filled circles at the child end, indicating required features.</p>	 <p>A UML class diagram showing a package <b>LoginCredentials</b> containing a class <b>LoginCredentials</b>. The class has two public attributes: <b>+login: String</b> and <b>+password: String</b>.</p>
 <p>A feature construction diagram for <b>RegistrationInfo</b>. The root node is <b>RegistrationInfo</b>. It has two child nodes: <b>ShippingAddress (String)</b> and <b>BillingAddress (String)</b>. Both child nodes are connected to the root by dashed lines with open circles at the child end, indicating optional features. Multiplicity values of <b>0..1</b> are shown next to each child node.</p>	 <p>A UML class diagram showing package merging. A package <b>PRegistrationInfo</b> contains a class <b>RegistrationInfo</b>. Two other packages, <b>PShippingAddress</b> and <b>PBillingAddress</b>, are merged into <b>PRegistrationInfo</b>. <b>PShippingAddress</b> contains a class <b>RegistrationInfo</b> with an attribute <b>+ShippingAddress: String</b>. <b>PBillingAddress</b> contains a class <b>RegistrationInfo</b> with an attribute <b>+BillingAddress: String</b>. Red arrows labeled <b>&lt;&lt;merge&gt;&gt;</b> point from the sub-packages to the parent package.</p>
 <p>A feature construction diagram for <b>CheckOut</b>. The root node is <b>CheckOut</b>. It has three child nodes: <b>ShippingOptions</b>, <b>PaymentOptions</b>, and <b>TaxationOptions</b>. <b>ShippingOptions</b> and <b>TaxationOptions</b> are connected to the root by dashed lines with open circles at the child end, indicating optional features with multiplicity <b>0..1</b>. <b>PaymentOptions</b> is connected to the root by a solid line with a filled circle at the child end, indicating a required feature with multiplicity <b>1..1</b>.</p>	 <p>A UML class diagram showing package merging and associations. A package <b>PCheckOut</b> contains classes <b>CheckOut</b>, <b>PaymentOptions</b>, and <b>TaxationOptions</b>. <b>CheckOut</b> has a 1-to-1 association with <b>TaxationOptions</b>. Another package <b>PShippingOptions</b> is merged into <b>PCheckOut</b> and contains a class <b>ShippingOptions</b>. Red arrows labeled <b>&lt;&lt;merge&gt;&gt;</b> point from <b>PShippingOptions</b> to <b>PCheckOut</b>. Multiplicity values of <b>1</b> are shown at the association ends.</p>
 <p>A feature construction diagram for <b>CheckOutType</b>. The root node is <b>CheckOutType</b>. It has two child nodes: <b>Registered</b> and <b>Guest</b>. Both child nodes are connected to the root by dashed lines with open circles at the child end, indicating optional features with multiplicity <b>1..1</b>.</p>	 <p>A UML class diagram showing package merging and inheritance. A package <b>PCheckOut</b> contains a class <b>CheckOutType</b> and a class <b>CheckOut</b>. <b>CheckOutType</b> has a 1-to-1 association with <b>CheckOut</b>. Two other packages, <b>PRegistered</b> and <b>PGuest</b>, are merged into <b>PCheckOut</b>. <b>PRegistered</b> contains a class <b>CheckOutType</b> with an attribute <b>Registered</b>. <b>PGuest</b> contains a class <b>CheckOutType</b> with an attribute <b>Guest</b>. Red arrows labeled <b>&lt;&lt;merge&gt;&gt;</b> point from <b>PRegistered</b> and <b>PGuest</b> to <b>PCheckOut</b>. Multiplicity values of <b>1</b> are shown at the association ends.</p>

**Table 3 (cont. of 2)** The basic structural features and their translation to package combination of class diagrams



The same argumentation can be used if we want to find behavioral patterns. The obvious solution is to use as well the package merge mechanism. We base the UML behavioral model in use case diagrams, with intensive use of include and extend relationships. Mandatory behavior is modeled with included use cases and optional behavior with extension use cases. But optional behavior has, as in structural models, two meanings. First, the details of payment steps can be different in an implemented application of the product line because it is possible to pay by cheque or credit card (and then we use the extend relationship). Second, any application requires always the credit card payments but not the cheque possibility (this behavior is in an extension use case inside an optional package). This approach guides us to the Table 4 patterns (we have included only the four basic patterns: optional/mandatory include/extend version of the common situations).

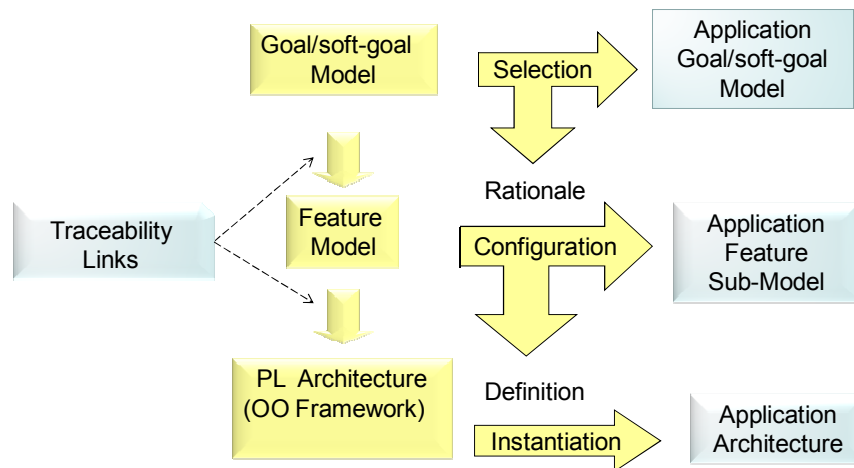
**Table 4** The basic behavioral features and their translation to package combination of UML use case diagrams

Feature Construction	Use Case diagrams (package merge variant)
	
	



## 4 MDE based transformations of feature patterns

According to Figure 9, in a product line approach and in the derivation of concrete application phase, several transformations are required. The first step includes the selection of the optimal combination of goals and soft-goals that respond to the specific customer preferences. This selection leads by traceability links to a (possibly partial or staggered) configuration of the product line feature model. The next step is the derivation of the application architecture: The initial application domain model obtained from the feature model and the asset base will be manually completed and optimized. All the models in Figure 9 can be considered as PIMs. Out of the scope of this Figure, the translation of an application (platform independent) model to specific platforms (.NET or EJB) is a conventional PIM to PSM transformation that can be defined using MDE transformation tools.



**Fig. 9** Transformation between the models of a product line

To enable this derivation process, the traceability links between the diverse PL models must be carefully established. The feature patterns transformation focused in this article can be automated, following the MDE paradigm. The method we have chosen is based on the meta-model mapping approach [9]. The work consists of defining a set of transformations between the patterns in the feature models and the architectural UML constructions, based in the results of the previous section.

The way to define a transformation is to find an element (or more exactly a combination of elements that satisfy certain constraints) of the feature meta-model and give one or several equivalences in the UML meta-model. This implies that an annotation is needed in the feature model to indicate which of the two possible transformation mechanisms is applied (structural, behavioral or both). As we need a precise definition of the meta-models and the target meta-model is UML, the first step is to select the features meta-model. Several authors have specified different meta-

models using MOF. We have evaluate these meta-models as the election influences greatly the transformation process and, finally the one proposed by Czarnecki *et al.* in [10] has been selected because the simplicity of the related transformation. In this approach, the distinctive property of the relationships is the cardinality. Figure 10 shows the above-mentioned meta-model and Figure 11 the UML target meta-model. In the meta-model of Figure 10, the relationships are implicit and the source of the transformation must be the cardinality attribute of the features and group of features. In previous work [22], we have defined and implemented the structure of packages that represent the architectural vision of the product line. The transformation is now complete, filling the packages with classes and attributes.

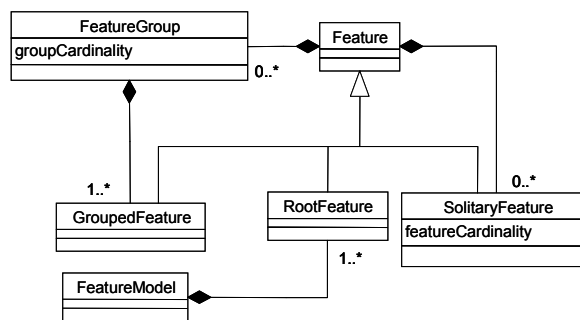


Fig. 10 Feature simplified meta-model, adapted from Czarnecki *et al.* [10]

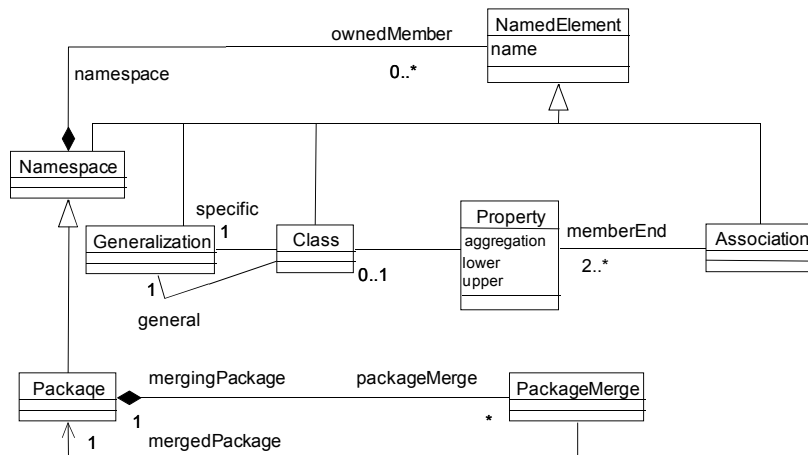


Fig. 11 partial UML meta-model (structural concepts)

The transformation implies:

- a) Transform the Feature model into a UML model.
- b) Transform each RootFeature into a root Package and an included class with the same name as the feature.
- c) Transform each mandatory FeatureGroup (i.e., with minimum cardinality greater than zero, as in examples 4 and 5 of Table 1) into a super-class of the set of classes generated by its owned GroupedFeature instances and generate an association with the class generated from the owner Feature with multiplicity equal to the groupCardinality.
- d) Transform each optional FeatureGroup (i.e., with minimum cardinality equal to zero, as in example 3) into a package merged with the previous package. Then, inside this package, create a super-class of the set of classes generated by its owned GroupedFeature instances and an association with a duplicate of the class generated from the owner Feature with multiplicity equal to the groupCardinality.
- e) Transform each mandatory GroupedFeature into a class that specializes the class generated by the owner FeatureGroup (*CreditCard* example).
- f) Transform each optional GroupedFeature into a package merged with the previous package. Then, inside the new package, create a class that specializes in a duplicate of the class generated by the owner FeatureGroup (as in the *DebitCard* example)
- g) Transform each mandatory SolitaryFeature typed as FEATURE into a class and associate it with the class generated from the owner feature with multiplicity equal to the featureCardinality (*TaxationOp* in example 3).
- h) Transform each mandatory SolitaryFeature (all the subtypes except FEATURE) into a typed attribute and assign this attribute to the class generated from the owner feature (*Login* in example 1).
- i) Transform each optional SolitaryFeature (all the subtypes) into a package merged with the previous package. Then:
  - a. Inside the new package, transform the SolitaryFeature typed as FEATURE into a class and, additionally, associate this class with a duplicate of the class generated from the owner feature with multiplicity equal to the featureCardinality (*ShippingOptions* in example 3).
  - b. Inside the new package, transform the SolitaryFeature with type other than FEATURE into an attribute and assign this attribute to a duplicate of the class generated from the owner feature (*ShippingAdresss* in example 2).

The strategy is based in the three subtypes of Feature. The root of every tree in a Feature model (RootFeature) is transformed in a class and a recursive transformation of Solitary Features and Feature Groups linked to every feature is carried out. The presence of a group implies a class associated to the parent feature that is specialized into several subtypes (one per alternative feature). Figures 12 and 13 show the graphically expressed transformations (using the QVT syntax [28]) of a Model Feature into a UML structural model. A similar set of transformations definition has

been built in order to produce in parallel the use case package diagrams. In fact, the transformation is most interesting if we consider that the package structure obtained (and completed by the designer) can be used to automatically derive the application model by selecting the desired goal/features, as mentioned above. This possibility compensates the overcharge of complexity of the goal/feature traceability management in the architectural model.

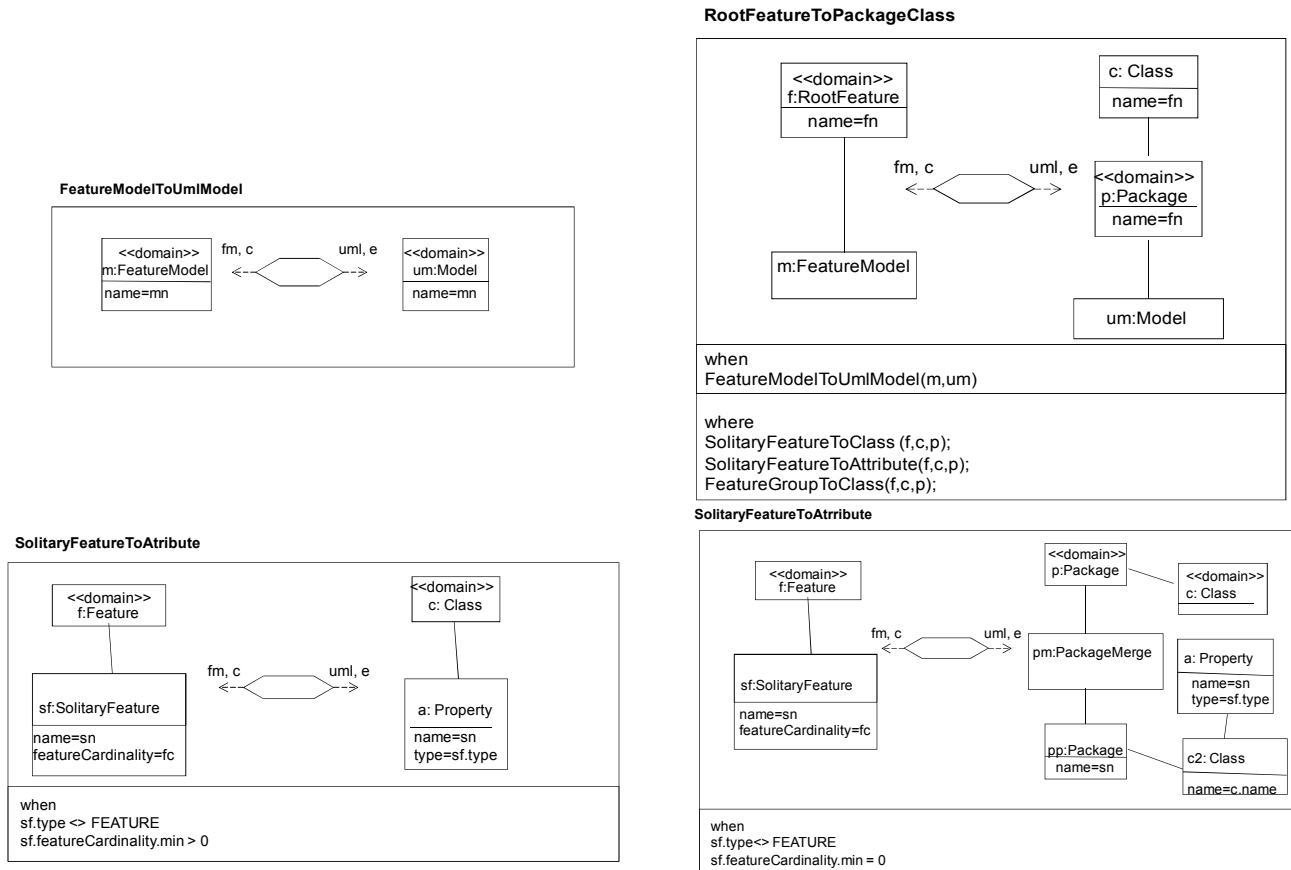
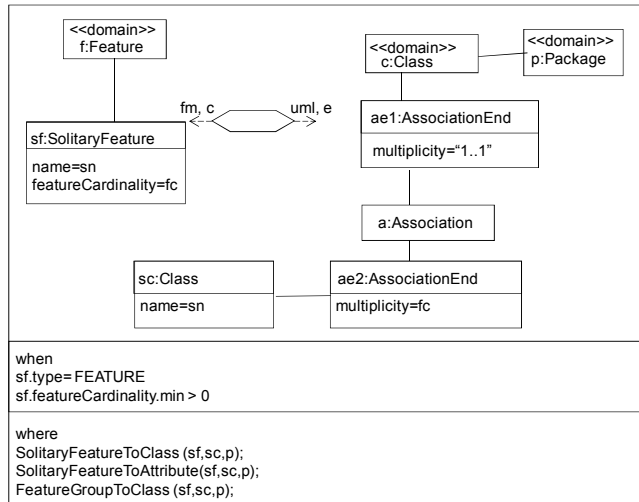
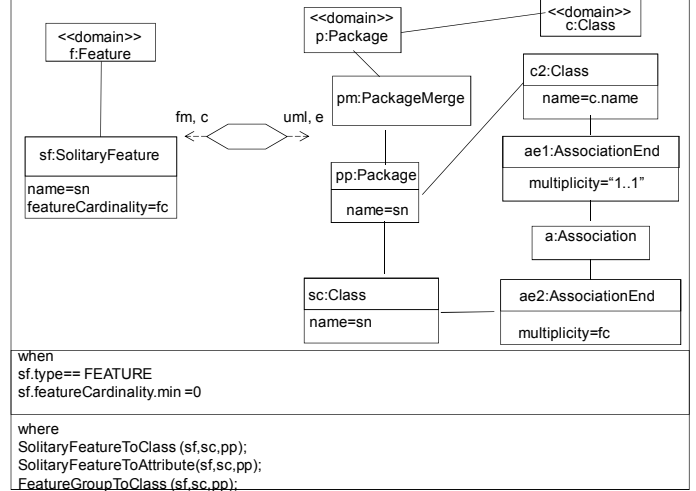


Fig. 12 Transformation definition of a Feature model into a UML/XMI model

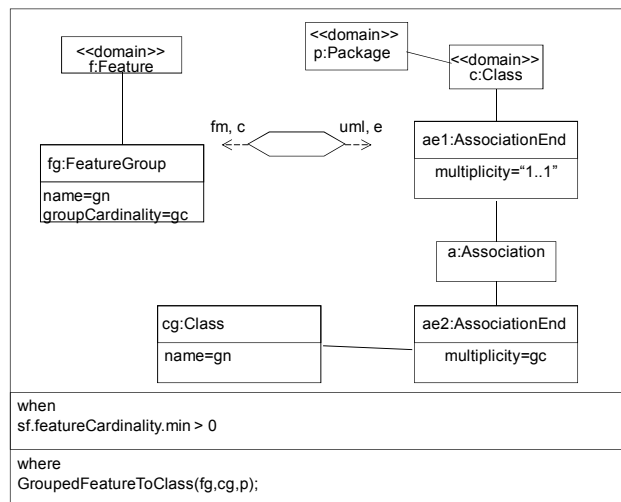
**SolitaryFeatureToClass**



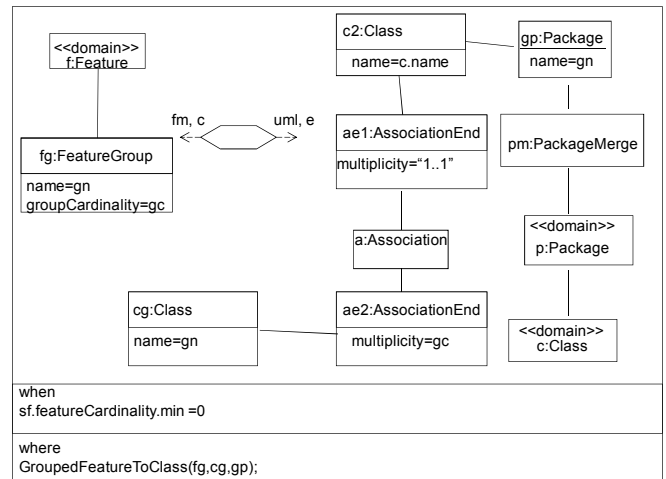
**SolitaryFeatureToClass**



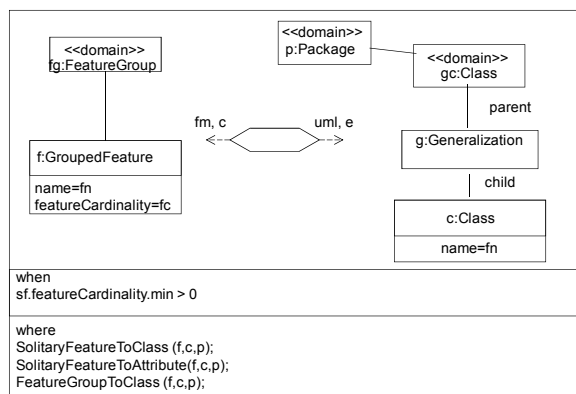
**FeatureGroupToClass**



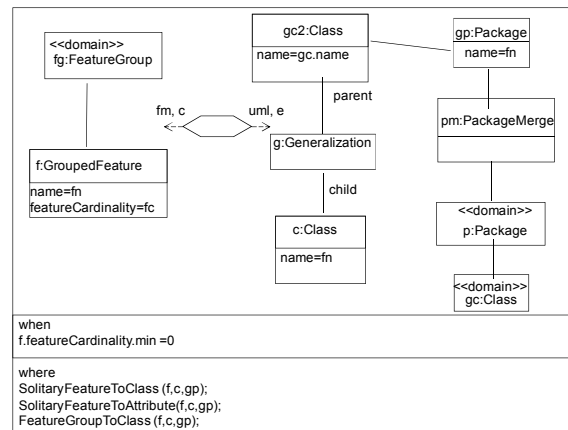
**FeatureGroupToClass**



**GroupedFeatureToClass**



**GroupedFeatureToClass**



**Fig. 13** Transformation definition of a Feature model into a UML/XMI model

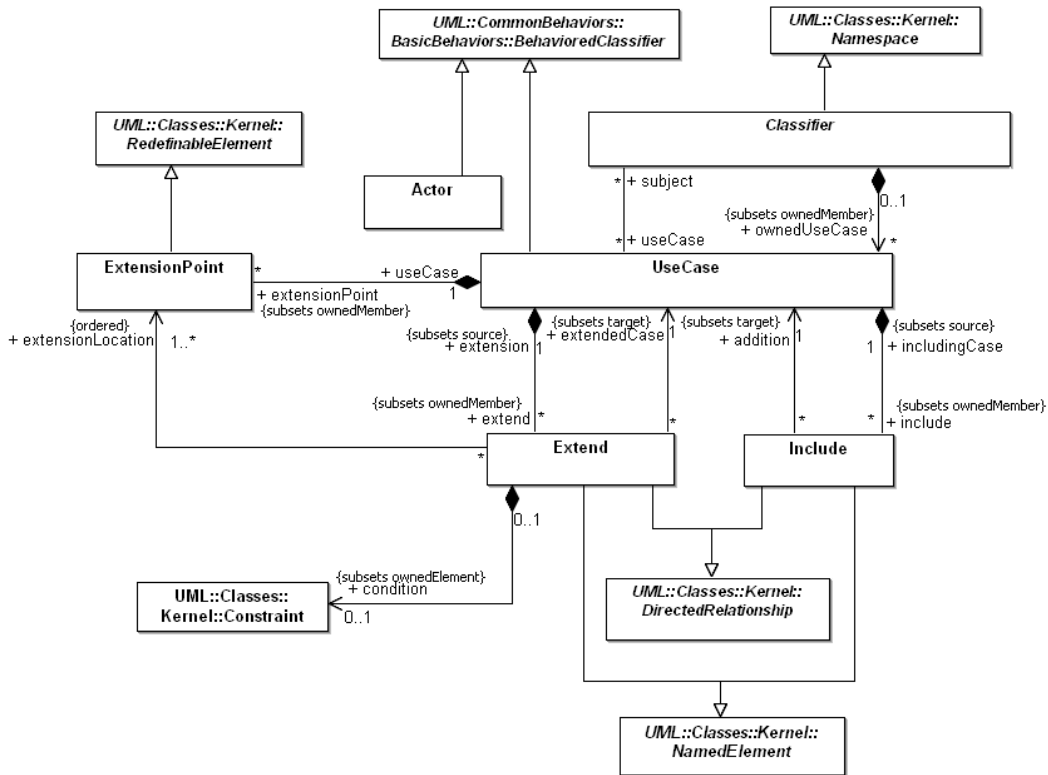


Fig. 14 partial UML meta-model (use case concepts)

A parallel transformation, from features into packages, use cases and extend/include relationships can be defined. The transformation consists of:

- Transform the Feature model in a UML model.
- Transform each RootFeature into a root Package and one included use case with the same name of the feature
- Transform each mandatory FeatureGroup (minimum cardinality greater than zero) into an included use case and generate an include relationship with the use case generated from the owner Feature.
- Transform each optional (minimum cardinality equal to zero) FeatureGroup into a package merged with the previous package. Then inside this package, create an included use case and generate an include relationship with a duplicate of the use case generated from the owner Feature.
- Transform each mandatory GroupedFeature into a use case that extends the use case generated by the owner FeatureGroup

- f) Transform each optional GroupedFeature into a package merged with the previous package. Then, inside the new package, create a use case that extends a duplicate of the use case generated by the owner FeatureGroup.
- g) Transform each mandatory SolitaryFeature typed as FEATURE into an included use case and an included relationship with the use case generated from the owner feature.
- h) Transform each optional SolitaryFeature typed as FEATURE into a package merged with the previous package. Then, Inside the new package, transform the SolitaryFeature into an included use case and, and include relationship with a duplicate of the use case generated from the owner feature

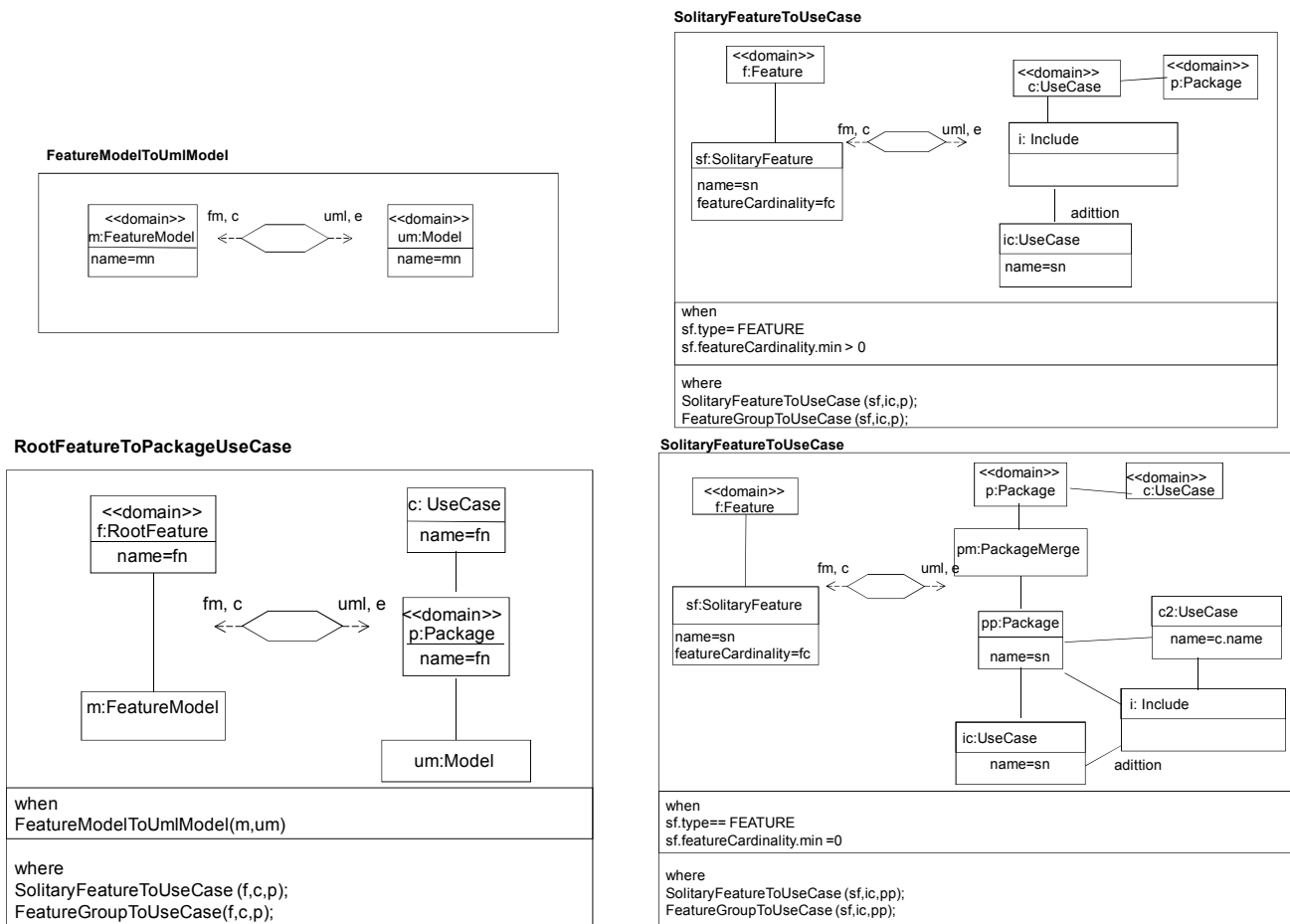
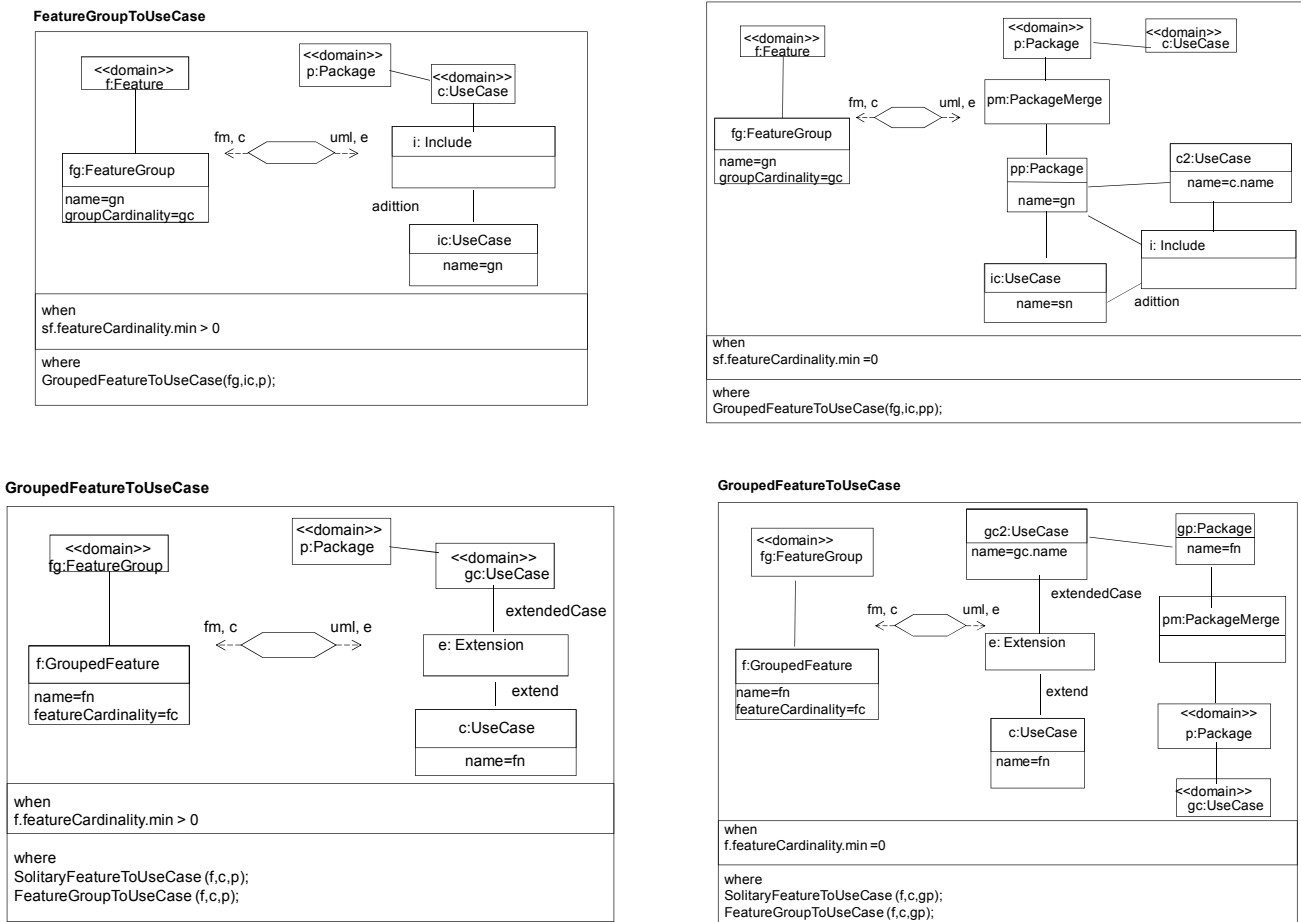


Fig. 15 Transformation definition of a Feature model into a UML/XMI model



**Fig. 16** Transformation definition of a Feature model into a UML/XMI model

The strategy is based in the three subtypes of Feature as in the structural transformations. The features of type different from FEATURE are ignored. Figures 15 and 15 show the graphically expressed transformations (using the QVT syntax [28]) of a Model Feature into a UML behavioral model.

Concerning the implementation details, a partial implementation, using a XML style sheet, is given in [22]. We use a combination of tools based on the Eclipse platform (the feature and UML models and the feature configuration utilities) and Microsoft proprietary tools (the goals analysis tools and Visual Studio for the C# code and the package configuration). The connection between them is achieved using intermediate XML files that can be automatically generated and manually completed. Recently we have finished the development of a specific domain language functionally using MS DSL tools equivalent to the *fmp* eclipse plug-in, in order to integrate all the phases of the process, from goals analysis to the application package configuration inside the MS Visual Studio platform.



## 5 Automated Transformations using FMT plug-in for Visual Studio.

We have developed a specific domain language functionality equivalent to the fmp eclipse plug-in in order to integrate all the phases of a development process from goal analysis to the application package configuration inside the MS Visual Studio Platform.

This tool introduces product line as one of the project types provided by the development platform. The interface and underlying meta-model of the FMT is similar to the fmp eclipse plug-in and compatible with it, allowing the direct import of fmp models. The advantages of this FMT are direct integration into the Visual Studio IDE and the possibility of visual representation and manipulation of features and mutex/require constraints. As additional benefits, the package structure of the product line and configuration files can be directly generated.

The developed tool allows the user to define and configure a feature model in order to obtain a package model as a main framework of the product line. The package model is obtained using a code based transformation using XMI in order to get the diagram using a CASE tool. We apply the previously defined transformation to obtain a complete package model, including classes, attributes and relationships.

We show the functionality of this application with the following example based on the thesis of Sean Quan Lau [37]. We only consider a subset of the complete feature model presented in this thesis. The following figure shows the feature model of this e-commerce example designed with our FMT tool for Visual Studio Platform.

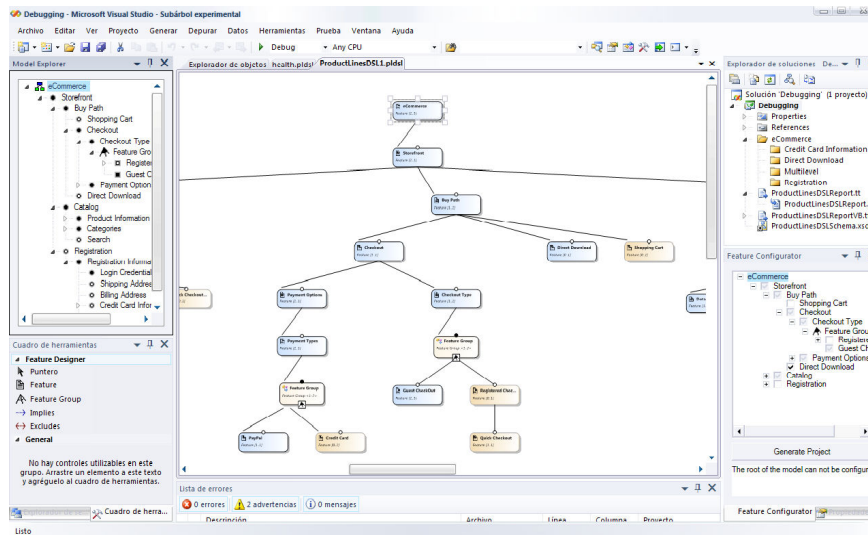
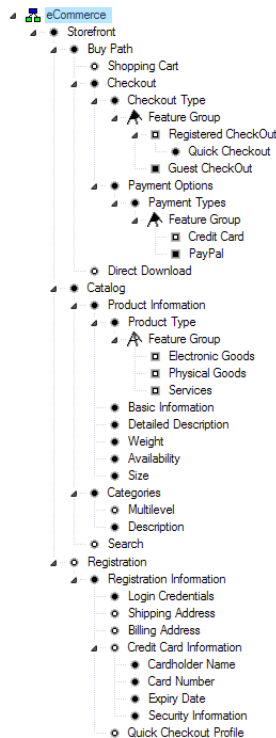


Fig. 17 Feature Modeling Tool.



**Fig. 18** Ecommerce Feature Model.

Once we have finished the definition of our feature model as is shown above, our tool can do automatically the transformation into a UML package model following the transformation definition mentioned in the previous chapter. The result of this transformation is a XMI document that we can open with a CASE tool, as we can see in the figure below (Figure 19).

The transformation has successful results. All mandatory features have been transformed into classes in the corresponding package. Typed leaf features have been transformed into attributes of a class. And finally, optional and group features have been transformed into packages connected with a merge relationship.

According to the transformations definition in the previous chapter, this transformation is correct, but we want to determine if it is really useful in order to develop a product line specific product.

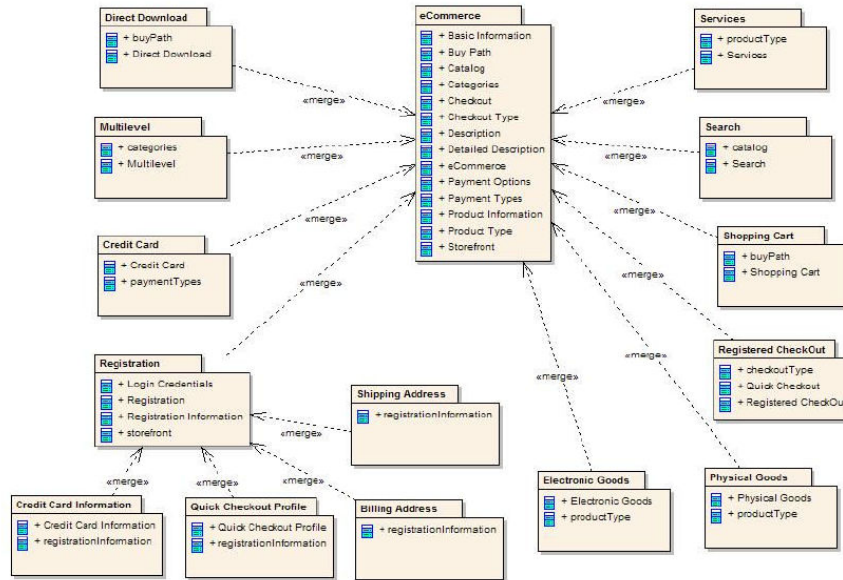


Fig. 19 Package Model obtained from a feature model using FMT.

In order to check the usefulness of this transformation we have compared this automated development of this e-commerce product line, with the manual development of the same product line that we have developed previously, with the following results:

	Manual Development	Automatic
<b>Packages created</b>	14	15
<b>Merge relationships created</b>	12	14
<b>Classes Created</b>	39	36
<b>Relationships created</b>	27	24
<b>Attributes created</b>	56	8

<b>Coincidence between packages</b>	81,25%
<b>Coincidence between merge relationships</b>	100%
<b>Coincidence between classes</b>	36,83%
<b>Coincidence between relationships</b>	16%
<b>Coincidence between attributes</b>	5,35%

<b>Useful Packages</b>	86,67%
<b>Useful Classes</b>	58,33%
<b>Useful Attributes</b>	100%

The automatic transformation has generated less classes and relationships than the manual example. The coincidence in the basic structure is almost a 100%, so it looks reasonable to use this automatic transformation to create the main framework to develop a specific product of the PL. The merge relationship between these packages has been generated successfully too.

The problem comes when if we want to do classes attributes and relationships transformations. The accuracy between the manual example and our automatic transformation is not very good. Only a 37% of the classes have been included in the manual development and the amount of attributes and relationships are even lower.

However, if we only consider the useful elements generated in the automatic transformation, we can see that the results are not so bad. The percentage of useful classes is almost 60%, i.e. 60% of the generated classes can be used or have been used in our manual development, which is a satisfactory result. The results of attributes are even better, because all attributes generated automatically have been used in the manual development.

We have proved the transformation into UML Use Case packages as it is shown in the previous chapter but the results are not as good as we expected. The correlation between features and use cases is not very clear yet and the percentage of useful elements in this transformation is very low. For that reason, we won't include this transformation functionality in the tool.

In conclusion, with the obtained results we can consider the use of the automatic transformation an advantage in order to develop a product line specific product, we can save a lot of time, and we can begin the development of the product as soon as possible.

## **6 Related Work.**

Schobbens *et al.* [30] have revised the diverse variant of feature diagrams, clarifying the differences and establishing a generic formal semantics. The influence of non functional requirements preferences in variant selection has been faced by several authors. The original FODA proposal uses the feature models for representing all the types of variability, functional and non functional [20]. Jarzabek *et al.* address the non functional requirements and feature relationships in the product line context [17]. They extend FODA with concepts of goal-oriented analysis. The proposed framework allows developers to record design rationale in the form of interdependencies among variant features and soft-goals. Both models are in the same level. Bosch [2] proposes an assessment method that addresses design decisions and non functional requirements in product-line development. In his approach, different profiles are used in relation to different "ilities" (usage profile for reliability or change profile for maintainability). Kazman *et al.* [21] proposed the SEI the architecture tradeoff analysis method (ATAM) for assessing the influences of architectural decisions on the quality attributes. Finally Yu *et al.* present in [35, 36] a model-driven extension to their Early Requirements Engineering tool (OpenOME) that generates an initial feature model from stakeholder goals.

Also the work devoted to relate feature constructions and architectural designs is abundant. Recent proposals express variability with UML models modifying or annotating these models. Structural, functional or dynamical models have been used. Some authors have proposed explicitly representing the functional variation points by adding annotations or changing the essence of the use case diagrams. For example, Von der Maßen et al. [34] propose using new relationships ("option" and "alternative") and the consequent extension of the UML meta-model. John and Muthig [18] suggest the application of use case templates to represent the variability in product lines, using stereotypes, though they do not distinguish between optional, alternative or obligatory variants. On the other hand, Halman and Pohl [15] defend the modification of the use case models to represent the variation points orthogonally (using a triangle symbol with different annotations). In all these cases the original UML model is modified to obtain the desired purpose.

Concerning structural models, either the mechanisms of UML are used directly (through the specialization relationship, the association multiplicity, etc.), as in the case of Jacobson [16]; or the models are explicitly annotated using stereotypes. The work of Gomaa [12] is an example of this approach, since it uses the stereotypes <<kernel>>, <<optional>> and <<variant>> (corresponding to obligatory, optional, and variant classes). Similarly, Clauß proposes a set of stereotypes to express the variability in the architecture models: <<optional>>, <<variationPoint>> and <<variant>> stereotypes designate respectively optional, variation points (and its subclasses), and variant classes [5].

The mapping between requirements and design has been always considered complex for several reasons (flexibility and adaptability of the product line, technology options, availability of resources, etc.). Sochos *et al.* provide an analysis on the product line methods and propose to strength the mapping between requirements and architecture modifying the feature models [31]. The disadvantage is the introduction of implementation characteristics in the requirements models.

Another solution, proposed by Czarnecki [7], consists of annotating the UML models with presence conditions, so that each optional feature is reflected in one or, more realistically, several parts of a diagram (which may be a class, an association, an attribute, etc. or a combination of elements). Although the class diagrams are the most used, the technique can be applied to any UML model, in particular the sequence or activity diagrams. Some tools are provided, such as an Eclipse plug-in for the definition and configuration of the feature model and an auxiliary tool to show the presence condition in UML templates.

In [11] MDA is presented as an approach to derive products in a specific type of product lines, configurable families. The authors' main idea is that a software system that is specified according to the MDA approach is a particular case of product line where the most characteristic variation point consists of products that implement the same functionality on different platforms. The choice for the alternative platforms is a variation point in such a product line. This variation point can be separated from the specification models and managed in the transformation definition itself. The main benefit of MDA compared to traditional development, is that the management of the platform variation point is handled automatically by the transformation step and is not a concern for the product engineer. However the final platform for a product is not the only variation point that needs to be managed in a product line. The various product

line members differ in both their functional and non-functional requirements. We think that our approach is more complete and flexible.

## **6 Conclusions and future work**

In this article, the possibilities provided by the combination of diverse modeling paradigms to represent and configure variability in a product line are discussed. The use of several models for representing the diverse facets of variability can improve the development of product lines. Other specialized techniques have shown better results for expressing different aspects of the requirement variability, while the feature model continues being the central piece of the puzzle.

The main contribution of the article is the identification of patterns in the feature models and the mapping of these patterns with the correspondent architectural diagrams. The feature patterns catalog allows the automated creation of traceability links between the product line feature and the architectural models, and consequently the productivity in product line development is improved. The final conclusion is positive as the combination of paradigms and the patterns catalog makes more straightforward the development process of the product line.

A set of transformations based in QVT are partially implemented to automatically obtain the UML and structural models, filling the package structures already implemented in our previous work. Work in progress include the development with of a specific domain language functionally that will integrate all the phases of the process, from goals analysis to the application package configuration.

As future work, an advanced vision is based more strictly on the MDE paradigm, automating most of the phases of product line development. First, the set of UML domain and behavior models are obtained using the feature pattern transformations (and manually completing these models). Then, the goal based configuration process yields a subset of packages that will be merged at conceptual level in a monolithic model (using existing MDE tools). The resulting (platform independent) model will be used as input to code generators tools. These tools are precisely intended to generate the platform specific models and the final code. We are evaluating some of the best known tools in order to assess the practical possibilities of this product line and MDE alliance.

## **Acknowledgements**

This work has been supported by the Junta de Castilla y León (project VA018A07).

## **References**

1. Antkiewicz, M., and Czarniecki, K. Feature modeling plugin for Eclipse. In OOPSLA'04 Eclipse technology exchange workshop (2004).

2. Bosch, J. “*Design & Use of Software Architectures. Adopting and Evolving a Product-Line Approach*”. Addison-Wesley. 2000.
3. Chastek, G., Donohoe, P., Kang, K. C., Thiel, S. “*product line Analysis: A Practical Introduction*”. Technical Report CMU/SEI-2001-TR-001 ESC-TR-2001-001, Software Engineering Institute (Carnegie Mellon), Pittsburgh, PA 15213
4. Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers 2000.
5. M. Clauß. Generic modeling using Uml extensions for variability. In Workshop on Domain Specific Visual Languages at OOPSLA 2001, 2001.
6. Clements, Paul C. and Northrop, Linda. “*Software product lines: Practices and Patterns*”. SEI Series in Software Engineering, Addison-Wesley. 2001.
7. K. Czarnecki, M. Antkiewicz, Mapping features to models: a template approach based on superimposed variants, In proc. International Conference on Generative Programming and Component Engineering (GPCE’05), LNCS 3676, Springer, pp. 422-437.
8. Krzysztof Czarnecki and Ulrich W. Eisenecker, “*Generative Programming: Methods, Tools, and Applications*”, Addison-Wesley, 2000
9. Krzysztof Czarnecki, Simon Helsen. “*Classification of Model Transformation Approaches*”. OOPSLA’03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.
10. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. Software Process Improvement and Practice, special issue on "Software Variability: Process and Management, 10(2), 2005, pp. 143 - 169
11. Sybren Deelstra, Marco Sinnema, Jilles van Gorp, Jan Bosch, “*Model Driven Architecture as Approach to Manage Variability in Software Product Families*”, in Arend Rensink (Editor), Model Driven Architecture: Foundations and Applications, CTIT Technical Report TR-CTIT-03-27, University of Twente, available in <http://trese.cs.utwente.nl/mdafa2003>
12. H. Gomma. Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, IEEE International Conference for Software Reuse (ICSR6), pages 89–99, June 2000.
13. González-Baixauli, B., Leite J.C.S.P., and Mylopoulos, J. “*Visual Variability Analysis with Goal Models*”. Proc. of the RE’2004. Sept. 2004. Kyoto, Japan. IEEE Computer Society, 2004. pp: 198-207.
14. Griss, M.L., Favaro, J., d’Alessandro, M., "Integrating feature modeling with the RSEB", Proceedings of the Fifth International Conference on Software Reuse, p.76-85. , 1998
15. Halmans, G., and Pohl, K., “Communicating the Variability of a Software-Product Family to Customers”. Journal of Software and Systems Modeling 2, 1 2003, 15--36.
16. Jacobson I., Griss M. and Jonsson P. “*Software Reuse. Architecture, Process and Organization for Business Success*”. ACM Press. Addison Wesley Longman. 1997.
17. Jarzabek, S.; Yang, B.; Yoeun, S., "Addressing quality attributes in domain analysis for product lines," *Software, IEE Proceedings* - , vol.153, no.2, pp. 61-73, April 2006
18. John, I., Muthig, D.: Tailoring Use Cases for product line Modeling. Proceedings of the International Workshop on Requirements Engineering for product lines 2002 (REPL’02). Technical Report: ALR-2002-033, AVAYA labs, 2002
19. Kang, K. C., Kim, S., Lee, J. y Kim, K. “*FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures*”. Annals of Software Engineering, 5:143-168. 1998.
20. K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. “*Feature-Oriented Domain Analysis (FODA) Feasibility Study*”. Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute (Carnegie Mellon), Pittsburgh, PA 15213

21. Kazman, R., Klein, M., and Clements, P.: 'ATAM: method for architecture evaluation', Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 2000
22. Miguel A. Laguna, Bruno González-Baixauli, José Manuel Marqués, Seamless Development of Software Product Lines: Feature Models to UML Traceability. Sixth International Conference on Generative Programming and Component Engineering (GPCE 07). Salzburg, Austria - oct 2007
23. Lau, S., "Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates", MASC Thesis, ECE Department, University of Waterloo, Canada, 2006.
24. Lee, K., Kang, K. C., Chae, W., Choi, B. W. "*Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse*". Software: Practice and Experience, 30(9):1025-1046. 2000.
25. S.J. Mellor, M. J. Balcer, "*Executable UML A foundation for the Model-Driven Architecture*", Addison Wesley Professional, 2002
26. J. Mylopoulos, L. Chung, and E. Yu. "*From object-oriented to goal-oriented requirements analysis*". Communications of the ACM, 42(1):31-37, Jan. 1999.
27. Object Management Group, "*MDA Guide Version 1.0*", 2003
28. Object Management Group and *QVT-Merge Group* , "*Revised submission for MOF 2.0 Query/View/Transformation version 2.0*" Object Management Group doc. ad/2005-03-02, 2005.
29. Schobbens, P.-Y., Heymans, P., and Trigaux, J.-C. Feature diagrams: A survey and a formal semantics. In RE, pp. 136-145. 2006.
30. Schobbens, P., Heymans, P., Trigaux, J., and Bontemps, Y. 2007. Generic semantics of feature diagrams. Comput. Netw. 51, 2, 456-479. Feb. 2007.
31. P. Sochos, I. Philippow, M. Riebisch. Feature-oriented development of software product lines: mapping feature models to the architecture. Springer, LNCS 3263, 2004, pp. 138-152.
32. van Lamsweerde, A. "*Goal-Oriented Requirements Engineering: A Guided Tour*", Proceedings of the 5 IEEE Int. Symp. on Requirements Engineering, 2001, pp: 249-262
33. A. van Deursen and P. Klint. "*Domain-Specific Language Design Requires Feature Descriptions*". Journal of Computing and Information Technology 10(1):1-17, 2002.
34. Thomas von der Massen, Horst Lichter, "RequiLine: A Requirements Engineering Tool for Software product lines". In Software Product-Family Engineering, PFE 2003, Siena, Italy, LNCS 3014 pp 168-180, 2003.
35. Y. Yu, A. Lapouchnian, S. Liaskos J. Mylopoulos and J.C.S.P. Leite. From goals to high-variability software design, pp. 1-16, In: 17th International Symposium on Methodologies for Intelligent Systems (ISMIS'08), 2008.
36. Y. Yu, A. Lapouchnian, J.C.S.P. Leite and J. Mylopoulos. Configuring Features with Stakeholder Goals. In: ACM SAC RETrack 2008
37. Sean Quan Lau Domain Analysis of E-commerce Systems Using Feature-Based Model Templates. Waterloo Ontario, Canada , 2006.