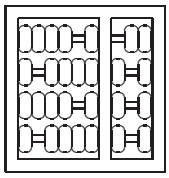

INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

A Systematic Approach for Architectural Design of Component-Based Product Lines

Ana Elisa de Campos Lobo
Patrick Henrique da Silva Brito
Cecília Mary Fischer Rubira

Technical Report - IC-07-33 - Relatório Técnico

November - 2007 - Novembro

A Systematic Approach for Architectural Design of Component-Based Product Lines*

Ana Elisa de Campos Lobo Patrick Henrique da Silva Brito
Cecília Mary Fischer Rubira

Abstract

Software product line is a systematic software reuse approach that promotes the generation of specific products from a set of core assets for a given domain, exploiting the commonalities and variabilities among these products. Feature modeling is one of the most accepted ways to represent commonalities and variabilities at the requirements phase. At the architecture design phase, product line architecture is an important core asset that should represent the common and variable parts in a product line. This technical report proposes an approach that supports the architectural design phase of component-based product line engineering. Our proposal for the conception of a product line architecture is based on the definition of variable architectural elements; and how they are derived from a given feature model and the interactions among features, providing a prescribed way to map features to the variable architectural elements composing the product line architecture. The main results of this work are a metamodel to define product line concepts extended with software architecture concepts, and a method for architectural design of product line engineering, to generate a product line architecture and instantiate it for a specific product. In order to evaluate the method, a case study was carried out, applying the approach to a product line in the mobile domain.

1 Introduction

Many software organizations develop and maintain a set of similar software products related to specific application domains. These products can be very similar, but they also present some differences in order to attend specific requirements from different clients. Software product line is an approach to improve development efficiency for families of software systems in a given domain. This approach facilitates large-scale reuse through a common set of core assets in a prescribed way [4].

The development process of a product line consists of two complementary subprocesses: product line engineering and application engineering. Product line engineering analyzes software products with regard to their commonalities and variabilities in order to build a

*Partially supported by CNPq, grant 484138/2006-5: Development and Evolution of Component-Based Software Product Lines.

reuse infrastructure that can be used to derive new similar products. Application engineering uses this infrastructure to instantiate particular software products [1]. Product line and application engineerings usually encompass three developing phases: requirements and analysis phase, design phase and implementation phase. The design phase is responsible for building the software architecture of the product line, which is coded in the implementation phase [8]. The software architecture is composed by architectural elements, which can be architectural components and architectural connectors. Components interacts with each other through interfaces, which represent the externally visible properties of components. [2].

In the context of software product lines, product line architectures should incorporate a variety of similar products in terms of their architectural elements. The more variable these architectural elements are, the more different products can be generated from them. Software variability [9] can be reached by delaying architectural design decisions, and it can be described through variation points and variants. A *variation point* is the place within an artifact where a design decision can be made, and *variants* are the design alternatives associated to this point [9]. A *decision model* describes how a variant is related to a certain high-level decision in a variation point in order to produce a specific product. The *product resolution model* represents the design decisions that were taken to generate a specific product [1]. Feature modeling is a well-known approach to represent commonalities and variabilities among products of a product line at the requirements and analysis phase [8, 5, 6]. However, very few investigations have been reported on providing a systematic mapping between the feature model and the architectural assets of a product line architecture [18, 17].

Therefore, a key factor to successfully implement an architectural product-line approach is to structure commonalities and variabilities into a product line architecture in terms of variable architectural components and connectors, and variable interfaces associated to variants of variation points. Our proposed solution focuses in the design phase of a product line engineering process, and it is based on an explicit mapping between the feature model and variable architectural elements which are architectural elements with variation points. This mapping is implemented by means of a decision model whose decisions should be chosen later on, during the design phase of the application engineering process.

Our solution presents at least three advantages. First, the explicit mapping between features and architecture prevents the phenomenon of feature scattering, where the implementation of a feature is spread in a number of architectural components. As a consequence, it is easier for one to trace how features evolution affects the product line architecture. Second, an explicit mapping between features and architectural elements can support automatic instantiation of products. Third, our solution also exploits features interactions in order to define new architectural elements and relationships among them, which are not identified if only the feature model is considered.

Our proposal consists of providing (i) a product line metamodel that unifies the concepts of software architecture and the concepts of variability and feature modeling in order to define variable architectural concepts, and (ii) a product line design method which describes how to generate and apply these concepts in a component-based product line engineering context. This method is a component-based method, and it is composed by two workflows: (i) a product line architecture design workflow to generate the product line architec-

ture for the product line, corresponding to the design phase of the product line engineering and (ii) an application architecture design workflow to generate a software architecture for a given product in the product line, corresponding to the design phase of the application engineering.

This technical report is organized as follows. Section 2 provides background information in feature modeling and feature interactions. Section 3 details the proposed approach, consisting of the product line design method (Section 3.1) and the product line metamodel (Section 3.2). In Section 4, the proposed approach is applied to a case study. Section 5 compares the proposed approach with related work. The last section presents some concluding remarks and directions for future work.

2 Background

2.1 Feature Modeling

A *feature* is a system property that is relevant to some stakeholder and it is used to capture commonalities and variabilities amongst systems in a product line [5]. *Feature modeling* is the activity of identifying commonalities and variabilities of the products of a product line in terms of features and organizing them into a feature model. A *feature diagram* represents a hierarchical decomposition of features including two types of relationships: aggregation and generalization. The aggregation relationship is used if a feature can be decomposed into a set of subfeatures, and the generalization relationship is used when a feature can be specialized into more specific ones with additional information [12].

Based on this structure, commonalities among all products of a product line are modeled as mandatory features, while variabilities among products are modeled as variable features. Variable features largely fall into three categories: optional, alternative, and multiple features. Optional feature is a feature that may or may not be present in a product. Alternative feature indicates a set of features, from which only one must be present in a product. Multiple feature represents a set of features, from which at least one must be present in a product. Besides the structural relationships between features, the feature model also can represent *additional constraints* between features. These constraints indicate which feature combinations are valid to generate a product in a product line. Some examples of constraints are *mutually dependency* (feature F "requires" feature G) and *mutually exclusion* (feature F "excludes" feature G).

Figure 1 shows a feature model for an Automated Teller Machine (ATM), using the representation proposed in [7], added by a representation for constraints. The root feature, ATM System, is composed by a mandatory features, User Identification and an optional feature, Balance. User Identification consists in alternative features Touch Screen and Card Reader. Balance is composed of multiple features Display and Printer, and the selection of Balance requires the presence of Card Reader.

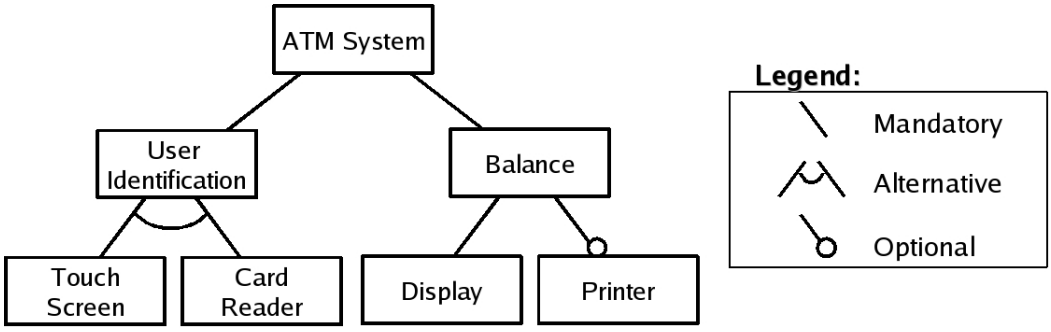

Figure 1: Feature Model for an Automated Teller Machine

2.2 Feature Interactions

A variable feature, if not properly designed, may affect a large part of product line assets. When features are independent from each other, each of them can be developed in isolation, and effects of a variation can be localized into the component implementing this feature. However, when features are not independent, a feature variation may cause changes to various components implementing related features [12]. Dependency constraints between features can be considered as a type of feature interaction. Although, a constraint is a kind of static feature interaction, because it only constrains the combination of features for a given product and it does not affect the operation of the components implementing this feature. On the other hand, feature interactions concern the interdependency between features, one affecting each other to achieve a common goal. There are many approaches and classifications for feature interactions [7, 12, 13]. For the purposes of this work, we adhere to the classifications proposed by [12], and we briefly present them in the following paragraph.

Feature interactions can be classified into at least three types of dependencies: usage dependency, modification dependency and activation dependency. In *Usage* dependency, a feature may depend on other features for its correct functioning or implementation. In *Modification* dependency, the behavior of a feature (a modifyee) may be modified by other feature (a modifier), while it is in activation. A feature should be active before it can provide its functionality to users. An *Activation* dependency occurs when the activation of a feature may depend on that of other feature.

3 The Proposed Approach

The proposed approach aims at providing a systematic way for deriving the product line architecture from a set of artifacts generated in the requirements and analysis phase, assuring an explicit mapping between features and the software architecture. It consists in a product line design method and a product line metamodel. The product line design method encompasses workflows for producing a product line architecture and instantiating the architecture for a given product. The method is based on four main concepts defined in the

product line metamodel: (i) the feature model; (ii) the software architecture model; (iii) the decision model; and (iv) the product resolution model. In order to define these concepts, the metamodel extends the concepts of software architecture with variable architectural components, and connects them to the feature modeling concepts, by means of the decision model and product resolution model.

3.1 The Product Line Design Method

The product line design method is composed by two complementary workflows: the *product line architecture design workflow*, responsible for developing the product line architecture and the decision model; and the *application architecture design workflow*, detailing the instantiation of the product line architecture to generate an application architecture for a specific product. Besides the representation of variabilities and features in an integrated way, the product line design method also details activities to identify and specify components and their interfaces, similar to the component specification phase of the UML Components process [3].

There are three artifacts, that should be provided as input for the method: (i) a feature model (Section 2.1); (ii) feature interactions, which extend the feature model detailing its dynamic aspects (Section 2.2); and (iii) a domain conceptual model, which represents the conceptual domain through either an entity-relationship diagram, or a UML class diagram. These artifacts can be constructed by a domain analysis method such as FODA [10].

3.1.1 Product Line Architecture Design Workflow

Figure 2 presents the activities of the product line architecture design workflow, which is used to create a product line architecture for a product line.

1-Architectural Design. This activity is responsible for providing the logical view of the software architecture. It can be done either by selecting a specific architecture for the domain or building the architecture using trade-off analysis of non-functional requirements [11]. For information systems, the method suggest the adoption of the layered architecture.

2-Identification of Feature Architectural Components. This activity is responsible for identifying the high-level components of the software architecture, whose interfaces are derived from the feature model provided. The activity is conducted through the five following steps:

1. Define at least one interface for each feature presented in the feature model. The quantity of interfaces depends on several factors (e.g. cohesion among components, architectural decision involving the trade-off between performance and maintainability);

2. Classify the interfaces according to the type of the respective feature that generated them: “optional” for optional features, “alternative” for alternative features, “multiple” for multiple features, and “normal” in other case;

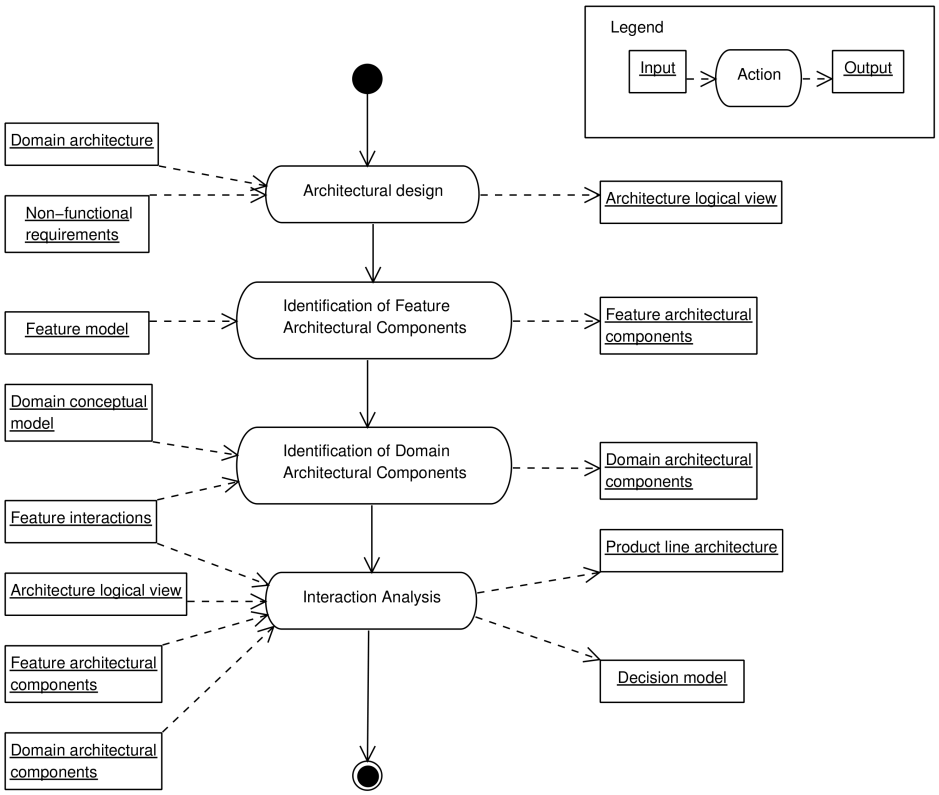

Figure 2: Activities of the Product Line Architecture Design Workflow

3. Group interfaces according to the cohesion among them. This step intend to simplify the software architecture in a disciplined way. A grouped interface is associated to a set of features, instead of a single feature;

4. Define architectural components for implementing the interfaces of the features. These components are called *feature architectural components*. The components with optional, alternative or multiple interfaces are classified as variable architectural elements.

5. Place the architectural components in the software architecture. This step is directly influenced by the software architecture defined in the Activity 1.

3-Identification of Domain Architectural Components. This activity is responsible for identifying other components of the software architecture, whose interfaces are derived from the domain conceptual model and the interaction among features in the feature model. The activity is composed by the following steps:

1. Select the main entities of the domain conceptual model. They are the ones needed by the features presented in the feature model.

2. Define at least one interface for each main entity of the domain conceptual model. The quantity of interfaces depends on several factors (e.g. cohesion among components, architectural decision involving the trade-off between performance and maintainability);

3. Classify the interfaces according to the type of the respective feature participating from the interaction that generated them: "optional" for optional features, "alternative" for alternative features, "multiple" for multiple features, and "normal" in other case;

4. Group interfaces according to the cohesion among them. This step intend to simplify the software architecture in a disciplined way. A grouped interface is associated to a set of features, instead of a single feature;

5. Define architectural components for implementing the interfaces of the domain entities. These components are called *domain architectural components*. The components with optional, alternative or multiple interfaces are classified as variable architectural elements.

6. Place the architectural components in the software architecture. This step is directly influenced by the software architecture defined in the Activity 1.

4-Interaction Analysis and Architectural Configuration. This activity is responsible for analyzing the interaction among the architectural elements that have been defined. Analyzing the interaction among features, it is possible to define the configuration of the product line architecture. Since it constitutes the design of the software architecture, it can also demand the definition of architectural connectors. When connecting through optional or alternative interfaces, it is necessary to create a variable connector, which is a kind of variable architectural element. Since the feature interaction represents the dynamic aspects among features, associations in this model can be mapped as a dependency between the components that provide the respective interfaces.
5-Construction of the Decision Model of the Software Architecture. This activity is responsible for grouping all the variabilities of the software architecture, as well as the respective decisions involving features that have to be taken to resolve them. The activity is conducted through the three following steps:

1. Associate each non-normal interface (optional, alternative, multiple) to a variant of the same type. Each variant is related to a condition (feature or set of features, according to the steps in activities 2 and 3).

2. Group all the alternative variants of the same condition in a variation point. In the same way, group all the multiple variants of the same condition in a variation point. Associate each optional variant to a variation point;

3. Associate each variation point to a feature or set of features, creating a decision;

4. Build a decision model by grouping all decisions of the software architecture.

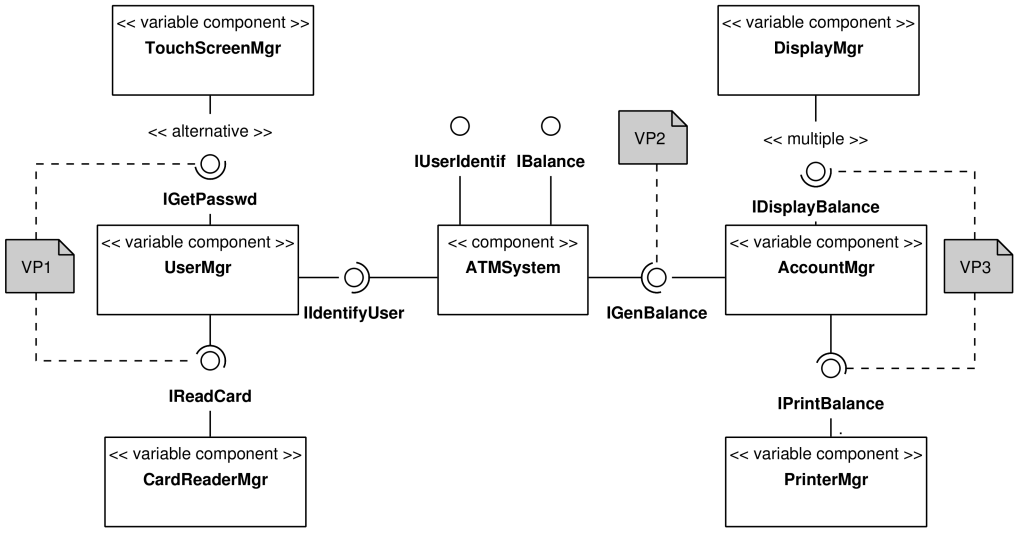

Figure 3: Product Line Architecture for ATM System

To illustrate how the mapping between feature model and software architecture works, we consider the feature model for the ATM described in Figure 1. In this example, we abstract the activities 1 and 3, and focus on activities 2, 4 and 5. For a more complete example, refer to the GoPhone case study (Section 4). During activity 2 (identification of feature architectural components), executing Step 1 and Step 2 we identify the interface *IIdentifyUser* associated to mandatory feature *User Identification*, the optional interface *IGenBalance* associated to optional feature *Balance*, alternative interfaces *IGetPassword* and *IReadCard* (associated to alternative features *Touch Screen* and *Card Reader*, respectively), and the multiple interfaces *IDisplayBalance* and *IPrintBalance* (associated to multiple features *Display* and *Printer*, respectively). In Step 3 and Step 4, we arrange these provided interfaces into the variable components *TouchScreenMgr*, *UserMgr*, *CardReaderMgr*, *DisplayMgr*, *AccountMgr* and *PrinterMgr*, and into the component *ATMSystem*.

During activity 4 (interaction analysis and architectural configuration) we construct the product line architecture, connecting required interfaces to provided interfaces through architectural connectors, providing the architectural configuration. In activity 5 (construction of the decision model of the software architecture) we construct the decision model. Alternative interfaces *IGetPassword* and *IReadCard* are associated to variants *Variant11* and *Variant12* of variation point *VP1*. Optional interface *IGenBalance* is associated to variant *Variant21* of variation point *VP2*, and multiple interfaces *IDisplayBalance* and *IPrintBalance* are associated to variants *Variant31* and *Variant32* of variation point *VP3*. Figure 3 shows the resulting product line architecture, represented as a UML component diagram extended with special stereotypes for alternative interfaces, multiple interfaces, optional

interfaces and variable components. A component with at least one optional, alternative or multiple interface is considered a variable component. Connectors are omitted in order to simplify the diagram. Variation points are indicated as comments associated to the related interfaces. Table 1 presents the resulting decision model aggregating all the design decisions associated to variants and variation points. In this table, the abbreviations *ri* and *pi* represent required interface and provided interface, respectively.

Table 1: Decision Model for ATM System

Component	VariationPoint	Type	Feature	Variant	Interfaces
UserMgr	VP1	Alternative	Touch Screen	Variant11	ri:IGetPasswd
			Card Reader	Variant12	ri:IReadCard
TouchScreenMgr	VP1	Optional	Touch Screen	Variant11	pi:IGetPasswd
CardReaderMgr	VP1	Optional	Card Reader	Variant12	pi:IReadCard
ATMSystem	VP2	Optional	Balance	Variant21	ri:IGenBalance
AccountMgr	VP2	Optional	Balance	Variant21	pi:IGenBalance
AccountMgr	VP3	Multiple	Display	Variant31	ri:IDisplayBalance
			Printer	Variant32	ri:IPrintBalance
DisplayMgr	VP3	Optional	Display	Variant31	pi:IDisplayBalance
PrinterMgr	VP3	Optional	Printer	Variant32	pi:IPrintBalance

3.1.2 The Application Architecture Design Workflow

Figure 4 presents the activities of the application architecture design workflow, which is used to instantiate a software architecture for a specific product, using the product line architecture generated in the product line architecture design workflow. Following, each one of its activities is presented in more details.

1-Generate the product resolution model. This activity is responsible for selection of the variable features to be present in a given product, and analyzing the design decisions in the decision model to generate a product resolution model. The activity is conducted through the three following steps:

1. Select the variable features for the product. For optional features, possible choices are whether the feature is present or not in the product; for alternative features only one subfeature can be selected for the product; and for multiple features at least one subfeature must be selected.

2. Validate the selection of features, assuring that the combination of features is valid for a product in the product line, and does not violate any constraint among features.

3. For each combination of selected features presented in the decision model, take the associated design decision connected to a variable architectural element, composing the product resolution model.

2-Instantiate the product line architecture. This activity is responsible for instantiating the product line architecture to generate a system component architecture for

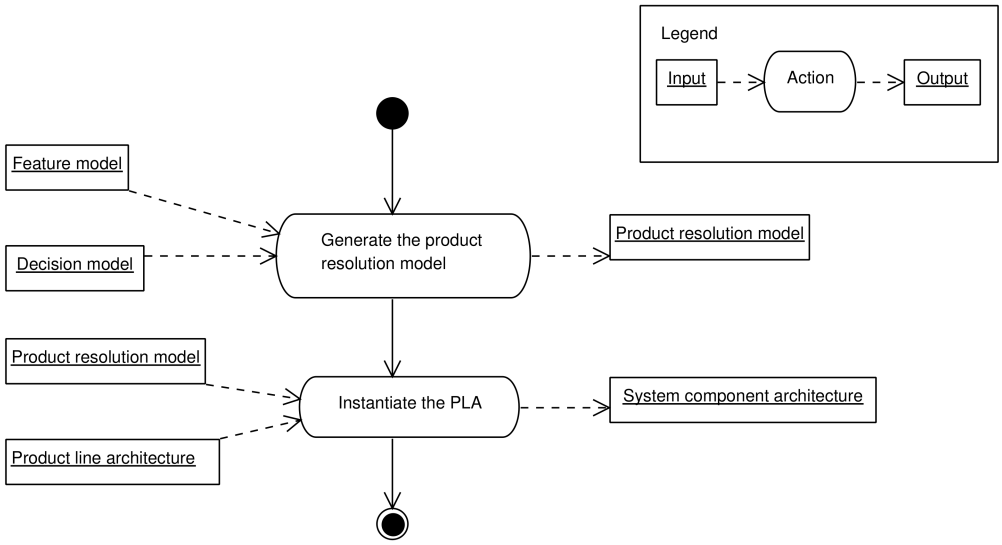

Figure 4: Activities of the Application Architecture Design Workflow

a specific product. This instantiation consists in using the product resolution model to resolve all the variabilities of the decision model of the product line architecture.

Using the example of the feature model of the ATM in Figure 1, executing Step 1 of activity 1 (generate the product resolution model), consider the following variable features were selected: *Card Reader*, *Balance* and *Printer*. As a mutually dependency constraint between *Balance* and *Card Reader* exists, *Balance* feature could be selected only because *Card Reader* feature was selected (Step 2). In Step 3, the decision model (Table 1) is used to generate the product resolution model, containing the interfaces *IReadCard* in components *UserMgr* and *CardReaderMgr*, the interfaces *IGenBalance* in components *ATMSystem* and *AccountMgr* and the interfaces *IPrintBalance* in components *AccountMgr* and *PrinterMgr*, related to each variable feature selected.

In activity 2 (instantiate the product line architecture) the product resolution model is used to instantiate the software architecture for the product from the product line architecture (Figure 3). As a result, the variabilities are removed from components *UserMgr*, *CardReaderMgr*, *AccountMgr* and *PrinterMgr*, and from their respective interfaces *IReadCard*, *IGenBalance* and *IPrintBalance*. On the other hand, the interfaces *IGetPasswd* and *IDisplayBalance*, and the components *TouchScreenMgr* and *DisplayMgr* are discarded. Therefore, the software architecture for this product consists in components *ATMSystem*, *UserMgr*, *CardReaderMgr*, *AccountMgr*, *PrinterMgr*, and the connectors and interfaces connecting them.

3.2 The Product Line Metamodel

The product line metamodel integrates the concepts of software architecture, feature modeling and software variability. This metamodel provides: (i) the definition of variable architectural elements, which are architectural elements with variation points; (ii) the composition of variable architectural elements to produce a product line architecture; (iii) the mapping between the feature model and the variation points into architectural elements by means of a decision model; and (iv) the definition of variability concepts to support our product line design method, presented in Section 3.1.

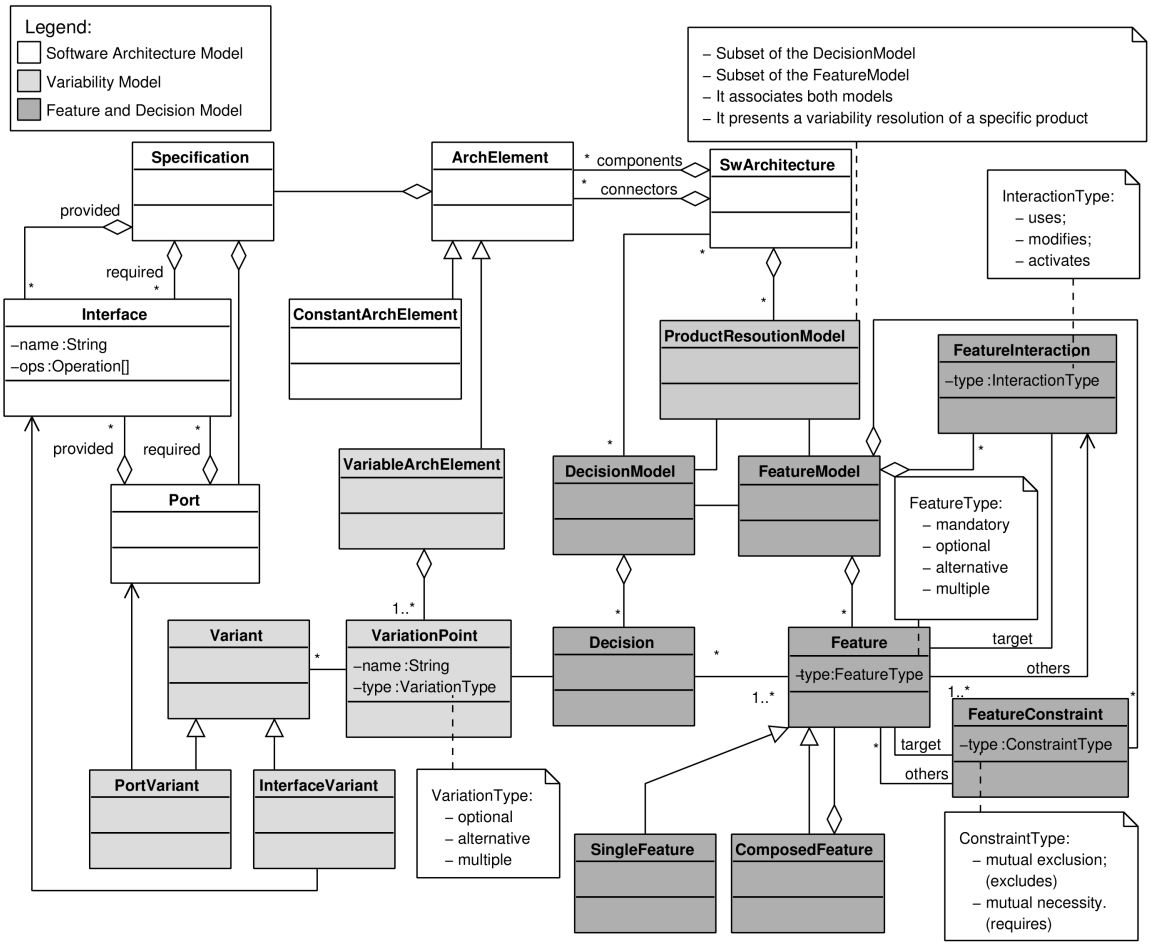

Figure 5: The Product Line Metamodel

Figure 5 presents the product line metamodel using UML notation to represent the concepts of software architecture (white classes) extended with variability concepts regarding a product line architecture (light gray classes). In order to support the development of product line applications, the metamodel relates the software architecture and variability

concepts with the concepts of feature modeling and decision modeling (dark gray classes). This mapping shows how a software architecture with variable architectural elements, the product line architecture, can be resolved to generate a software architecture for a specific product. The `ArchElement`s (architectural components or connectors) are specified through sets of provided and required interfaces, and a set of ports. A `Port` groups a set of provided and required interfaces.

To extend the architecture model with variability, we have defined a new type of architectural element (`VariableArchElement`), which has at least one `VariationPoint`. A `VariationPoint` is associated to a set of `Variant`s, and can be "optional", "alternative", or "multiple". Each `Variant` is either associated to an `Interface (InterfaceVariant)` or to a `Port (PortVariant)`.

In order to provide a way for constructing product line applications, a `VariationPoint` can be associated to a respective `Decision`, which refers to a `Feature`, being a bridge between the variability model and the feature model. All the `Decision`s that refer to a specific product line compose the `DecisionModel`. In the same way, all the `Feature`s related with the same product line compose the `FeatureModel`. A feature of a feature model can be considered mandatory or non-mandatory. Since the mandatory features are fixed parts of the feature model, they do not represent a design decision and consequently are not part of the decision model. Besides the set of `Feature`s, a `FeatureModel` also defines constraints among its features (Section 2.1). A `FeatureConstraint` is a static relationship among features. It can be: an "excludes" clause, to indicate mutual exclusion relation among features; or a "requires" clause, which indicates that a feature depends on other feature. In our metamodel, feature interactions also integrate the `FeatureModel`. A `FeatureInteraction` defines how a specific feature interact with other(s) of the same feature model (Section 2.2). A feature interaction can be: a "uses" clause, for usage dependency, a "modifies" clause, for modification dependency, or an "activates" clause, for activation dependency. A `Feature` can be a `SingleFeature` or a `GroupedFeature` (meaning aggregation or specialization), which has a set of children `Features`.

Besides the specification of product line architecture with the respective mapping to the respective decision and feature models, the proposed metamodel also supports the resolution of a decision model of a product line architecture for generating software architectures for specific products. These specific design decisions that were taken are represented through a `ProductResolutionModel`, which refers to how a specific product resolves the decision model of the product line architecture, relating the respective features affected by the decisions.

4 The GoPhone Case Study

This section describes a case study using an existent product line in order to demonstrate the viability of our solution. The selected context to conduct the case study was based on a technical report about the GoPhone Software Product Line (SPL) [15]. It was chosen for two reasons: the product line domain is well-known, and the availability of documentation and source code.

The GoPhone is a software product line applied to the mobile domain, implemented during the ViSEK project, aiming to provide material for researchers and practitioners. GoPhone is a hypothetical mobile phone company with a product portfolio of nine different types of phones, ranging from basic phones to sophisticated smart phones. Although having basic functionalities in common (such as making and receiving calls), these phones also have variabilities between each other, therefore they can be developed as a software product line. The GoPhone case study applies PuLSE and KobrA methods to generate a complete documentation and source code for the GoPhone SPL.

The next sections explain how the proposed method was applied to the GoPhone case study, as follows. Section 4.1 describes the pre-conditions to apply the product line method. Section 4.2 details the construction of the product line architecture, Section 4.3 details the instantiation of a software architecture for a specific product, and Section 4.4 evaluates the results of the case study.

4.1 Pre-conditions

The proposed method uses three artifacts as input: feature model, feature interactions model and domain conceptual model. Therefore, it is necessary to obtain these artifacts as a pre-condition to apply the method. The next paragraphs explains how these methods were obtained from the GoPhone case study, using other artifacts documented in [15].

The domain conceptual model (Figure 6) was constructed based on the domain structure diagram in [15]. It shows the main entities and the relationships between them, for the GoPhone SPL, as a UML class diagram.

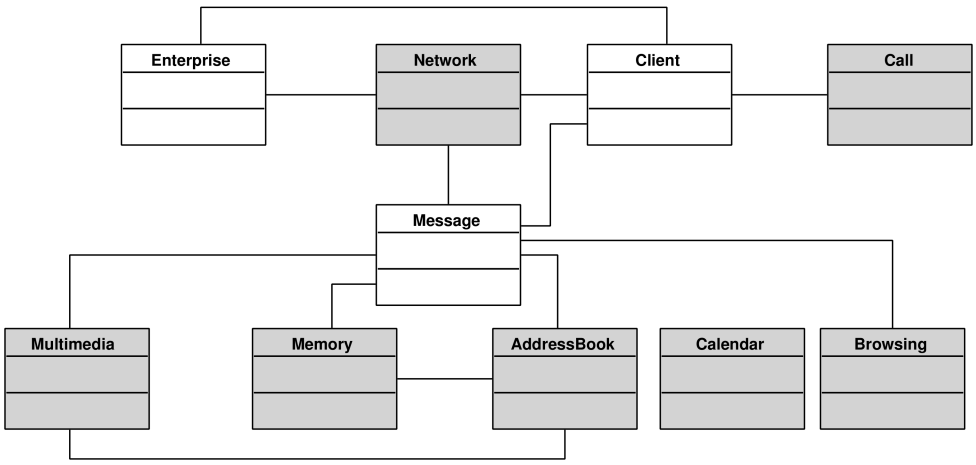

Figure 6: Domain Conceptual Model for GoPhone

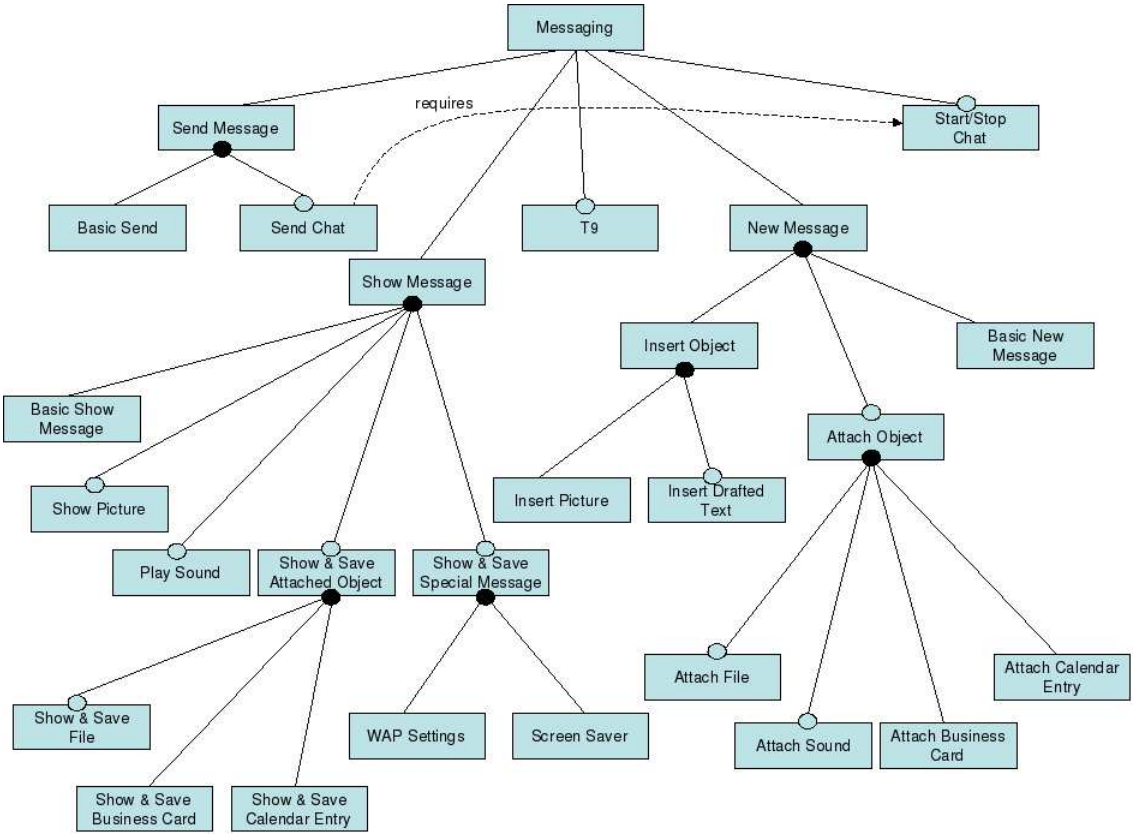

Figure 7: The Feature Model for GoPhone

The feature model for GoPhone SPL was constructed using the product map as a reference [15]. For limitations of space, this case study focused on the *Messaging* domain of GoPhone. Although, references to other domains of mobile phones is maintained to handle properly the interactions and dependencies between features. Considering the variabilities between products as shown in the product map, the feature model for the GoPhone product line is presented in Figure 7. This is a partial feature model, since only features related to the *Messaging* domain is presented, as explained before.

This feature model shows a root feature, `Messaging`, composed by five main subfeatures, three of them are mandatory features (`Send Message`, `Show Message`, `New Message`) and the other two are optional (`Start/Stop Chat` and `T9`). All the other subfeatures are derived from these main subfeatures. There is only one restriction, regarding the mutually dependency between subfeatures `Send Chat` and `Start/Stop Chat`. As a matter of comparison between products, the simpler version of GoPhone mobile is the GoPhone XS, and the most sophisticated one is GoPhone Smart. GoPhone XS has only the basic features (`Basic Send`, `Basic Show Message`, `Basic New Message`) and `T9`. On the other hand, GoPhone Smart has all multiple subfeatures for the basic features, including the optional subfeatures for chat (`Send Chat` and `Start/Stop Chat`), but does not have subfeature `T9`.

The feature interactions model was built based on the analysis of the interactions among features, using the domain structure diagram and the use cases described in [15]. The feature interactions diagram (Figure 8, using the representation suggested by Ferber [7]), shows the relationships between features of *Messaging* domain and between *Messaging* features and other domains (*Call Handler*, *Communication Handler*, *Multimedia*, *Memory Manager*, *Address Book*, *Calendar*, *Browsing* and *User Interface*). Even focusing on Messaging domain, this latter type of interactions is important to derive interfaces for domain architectural components (Section 3.1.1). Each domain is represented as a unique feature in the feature interactions diagram because their subfeatures were not detailed in the feature model.

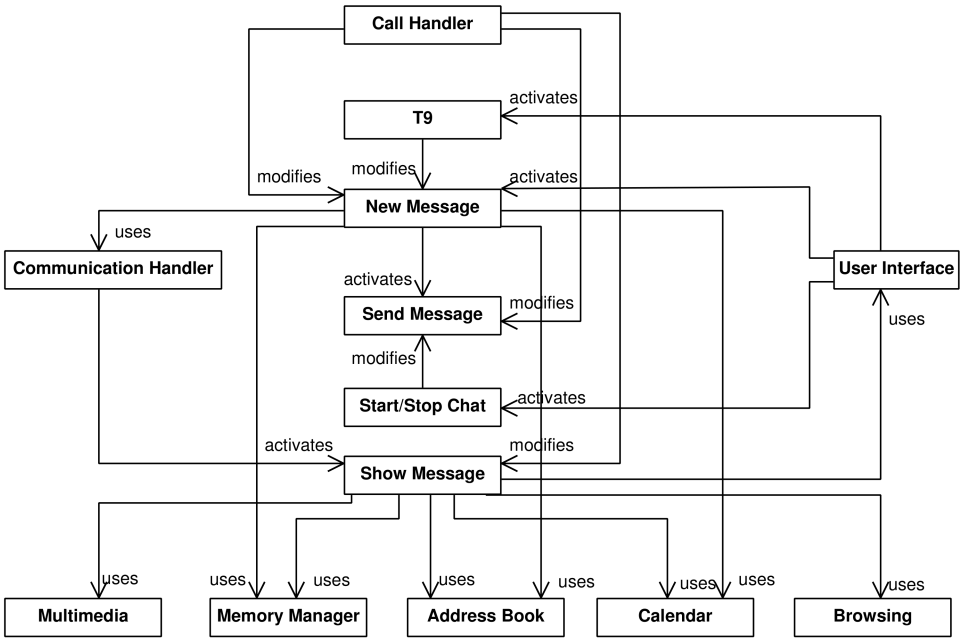

Figure 8: The Feature Interaction Model for GoPhone

4.2 Building the Product Line Architecture

According to our product line architecture design workflow presented in Section 3.1.1, the construction of a product line architecture starts with the selection of proper architectural styles or patterns [2] that fit the needs of the target systems' family. Since the GoPhone SPL is a kind of embedded hybrid system, with characteristics of both information and real-time systems, we have adopted a "relaxed" layered software architecture, which is presented in Figure 9 and combines characteristics of three-tier processing and direct access between some non-neighbor layers (object oriented style). Figure 9 shows the logical view selected for the product line architecture. It uses a relaxed layered style, since the relationships are

not restricted to adjacent layers. Our approach focuses on the business layers, in this case, *Feature Components Layer* and *Domain Components Layer*.

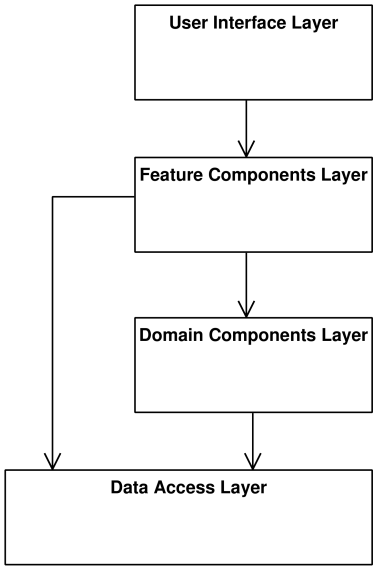

Figure 9: Architectural Logical View for GoPhone

After defining the high-level structure of the architecture (activity 1 of the workflow), we have to identify its internal elements in order to detail the logical view. The proposed method systematizes the identification of the architectural components, starting from their interfaces.

The first layer to be analyzed is the feature components layer, to identify feature architectural components (activity 2 of the workflow). Analyzing the feature model, we initially define one interface for each feature. After that, the discovered interfaces can be joined in a disciplined way in order to reduce the quantity of interfaces and simplify the model. One strategy suggested for joining interfaces is to create one interface for each low-level parent feature. For example, features `Send Message`, `Basic Send`, and `Send Chat` were joined into a single interface: `ISendMessage`. After discovering the interfaces, it is necessary to specify the variability of the interface, classifying it in accordance with the type of the respective feature: "optional", "alternative", "multiple" or "normal". Since `ISendMessage` is an obligatory service, it was classified as "normal". Finally, the designer has to create components for providing each discovered interface. It is important to stress that a component can provide any number of interfaces. Following the same example, for providing the `ISendMessage` interface we have defined the `MessageDispatcher` component.

Next, the domain components layer is analyzed to identify domain architectural components (activity 3 of the workflow). The goal of this activity is to identify the components that provides domain-related services. Due to the generic aspect of these components, they

are derived from the domain conceptual model, and are the most probable reused components. The first step for identifying the domain architectural components is to select a subset of entities of the domain conceptual model, which are relevant for executing the features of the feature model. This relevance can be analyzed answering the following question for each feature: "how an entity can be responsible for providing the informations required by the feature?" The feature interaction presented in Figure 8 supports the designer in this task. For example, analyzing Figure 8 and comparing with the domain conceptual model of Figure 6, it is possible to identify that there are seven entities which are relevant for in the context of messaging features; these elements are evidenced with a gray color in Figure 6. These elements can derive at least one provided interface, depending on the complexity of the feature interaction model provided by the user of the process. To exemplify the method with the GoPhone SPL, the entity `AddressBook` has support the identification of three required interfaces: `IBusinessCard`, `IPhoneNumber`, and `IEmail`.

Finally, as presented in the feature architectural components, the designer has to create components for providing each discovered interface. It is important to stress that a component can provide any number of interfaces. Following the same example, for providing the `IBusinessCard`, `IPhoneNumber`, and `IEmail` interfaces, we have defined the `AddressBookMgr` component. Figure 10 presents a partial product line architecture for GoPhone.

Analyzing the interaction among features, it is possible to know the way a component interact each other, and consequently define the configuration of the product architecture. In the feature interaction model for GoPhone (Figure 8) we verify various activation dependencies among *User Interface* feature and the *Messaging* features, as well as activation dependencies between *Messaging* features. This analysis demonstrates the necessity of creating at least one controller component, responsible to organize the sequence of operations in the resulting system, according to the operations showed in the feature interaction diagram. This controller component, `MessageController`, is not represented in the product line architecture (Figure 10) in order to simplify the diagram. During this activity, architectural connectors can also be defined, whenever necessary.

After the specification of the software architecture, all the variation points can be grouped to generate the decision model of the software architecture (activity 5 of the workflow). The decision model for GoPhone SPL is not presented here for limitations in space, but the principle is the same as shown in ATM System example, where each multiple and alternative interfaces will be grouped into variation points associated to the corresponding multiple and alternative features. Each optional interface is a variant and must be associated to a variation point related to the corresponding optional feature in the decision model. Therefore, the resulting decision model aggregates all design decisions in the product line architecture for GoPhone, and associates them to variable features in the GoPhone feature model.

4.3 Instantiating a Product

The application architecture design workflow (Section 3.1.2) aims to instantiate a software architecture for a given product of GoPhone. Take for example the simplest product, which

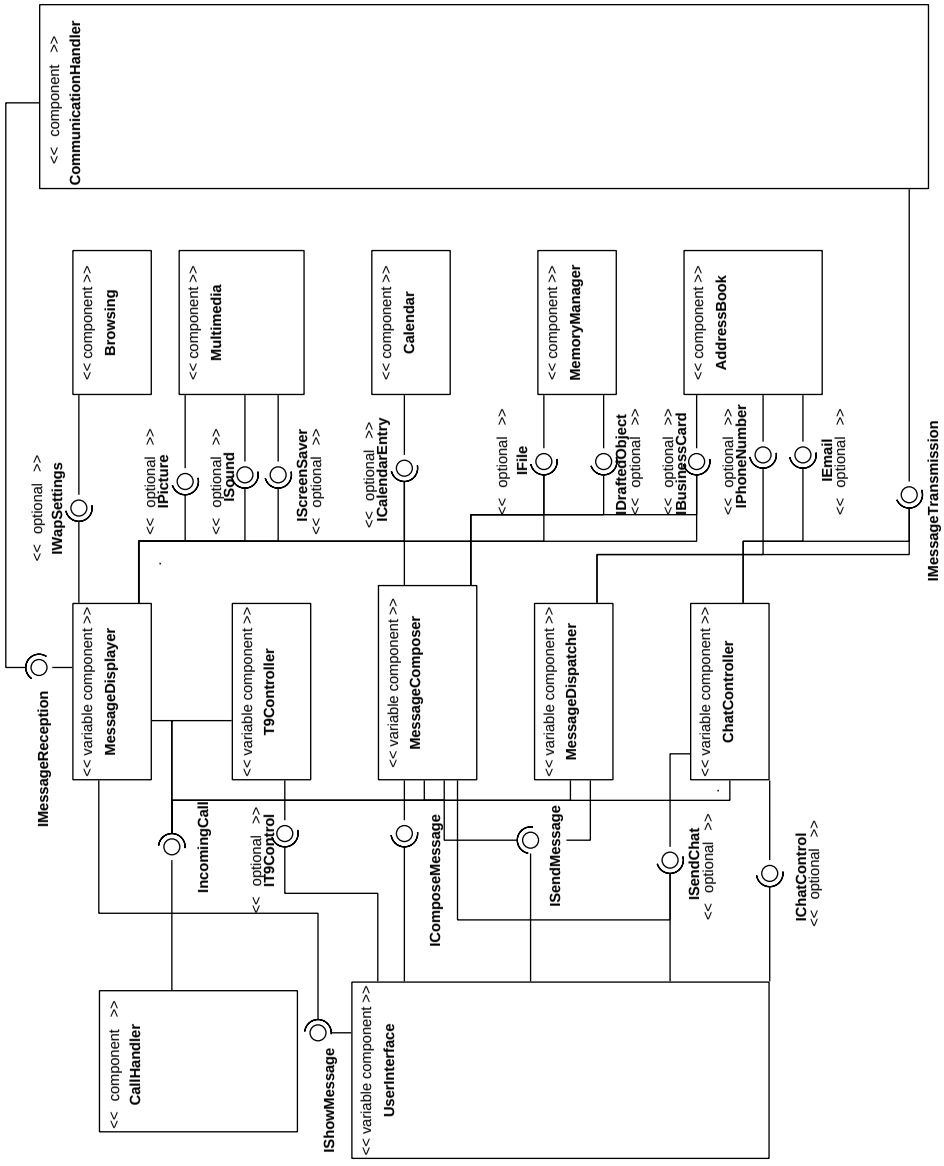

Figure 10: Product Line Architecture for GoPhone

is GoPhone XS. For this specific product, the only optional feature selected in the feature model is T9 (Figure 7). Besides that, only the mandatory features will be present in the product.

Applying the application architecture design workflow (Section 3.1.2) in the product line architecture for GoPhone (Figure 10), we conclude that only the optional interface `IT9Control` will be selected, all the others should be eliminated. As `IT9Control` interface is used, the component `T9Controller` is not variable anymore. In the same way, com-

ponent `UserInterface` is not a variable component either, since it resolved its variable interfaces keeping `IT9Control` interface and removing optional interfaces `ISendChat` and `IChatControl`. In the same way, variable components `MessageDisplayer`, `MessageComposer` and `MessageDispatcher` turn into components without variabilities. On the other hand, components which uniquely have optional interfaces and these interfaces were not selected must be removed from the resulting component architecture for GoPhone XS product. This is the case of components `ChatController`, `Browsing`, `Multimedia`, `Calendar`, `MemoryManager` and `AddressBook`. The final result of the application architecture design workflow is the architecture for GoPhone XS product, presented in Figure 11.

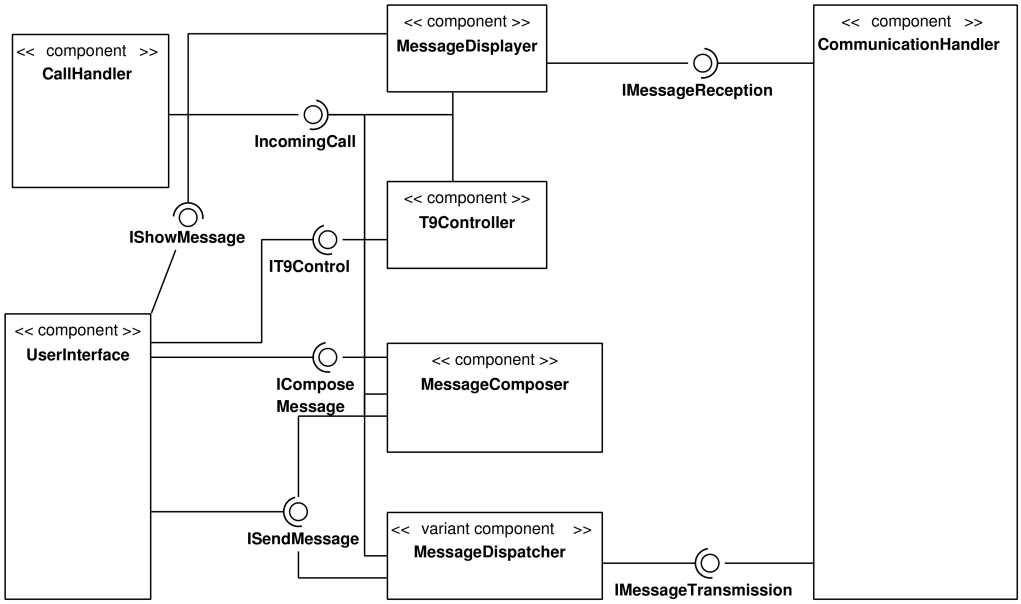

Figure 11: Architecture for GoPhone XS Product

4.4 Evaluation of the Case Study

This case study was conducted by two Ph.D. students and takes about 10 hours, including the time to generate the input artifacts (domain conceptual model, feature model and feature interaction model). Each activity was executed by one student and double-checked by the other one. All activities in the product line design method were executed, and all models were generated following the prescribed steps. The feature model contains 27 features, from which only 7 are mandatory features. The feature interaction model relates 13 features through 20 interactions. All relationships between features, either structural or operational, were considered to derive interfaces of the architectural components. However, as the method prescribes interface grouping, the number of generated interfaces is reduced.

The resulting product line architecture contains only 19 interfaces (some of them connecting more than two components), 12 of them being optional interfaces. These interfaces connects 7 components and 6 variable components.

Analyzing this results we can conclude that applying this method, the generated software architecture will contain at most the same number of components than features and at most the same number of interfaces than relationships among features. The ATM System example shows an extreme situation where these numbers are the same. It is more probable to have a smaller number of components and interfaces, as demonstrated the GoPhone case study.

Running the case study we conclude that the feature model must contain only functional features, and the feature modeling must be done carefully in order to the features can represent operations to derive interfaces. Another limitation of the method regards the architectural logical view, since it is focused in a layered architectural style. In spite of these limitations, we consider the case study successful and the method is validated, since the generated product line architecture is very similar to the real one, implemented by ViSEK project.

5 Related work

KobrA [1] is a component-based product line method which defines a set of extended UML models and a process to develop a product line. Kobra is a UML-based method which uses a set of models, mainly the use case model, to derive architectural elements. Like our approach, the decision model provides a mapping between features and the architecture. On the other hand, the method is not feature-oriented, therefore a feature can be represented in a number of use cases, causing the feature to be scattered among architectural elements.

Thiel and Hein [18] present a metamodel for product line variability. They extend the ANSI/IEEE 1471 recommended practice for architectural description to address the needs for documenting product lines by introducing extensions to model variability in features and in the architecture. They focus on mapping architectural variabilities in different architectural views, such as logical view, physical view, process view and deployment view. Although, they do not provide a systematic way to provide traceability between features and architectural variation points.

pure::variants [16] is a commercial tool designed for the development and deployment of software product lines. The tool is based on a set of models that are used to describe the problem domain and the solution domain, and the feature models plays a central role. The proprietary component model describes the implementation of the product. Each component in the model is a set of selectable parts of source code, which are chosen according to the feature selection. The developer should design these components carefully in order to generate the correct code.

The Feature-Architecture Mapping (FArM) [17] method provides a stronger mapping between features and the architecture. It is based on a series of transformations on the initial feature model. During these transformations architectural components are derived, encapsulating the business logic of each transformed feature and having interfaces reflecting the feature interactions. These transformations are applied iteratively, in regard to non-

architecture-related and quality features, architecture requirements, interacts relations and hierarchy relations. These transformations are applied to the initial feature model to generate intermediary feature models, in order to bring the structure of feature model closer to the architecture, allowing an easier mapping. Although this approach has similar results to ours, some transformations seem not always to be possible, as for quality and architectural requirements, without using any other artifact but the feature model.

6 Conclusions and Future Work

This technical report presented an approach that supports the architectural design phase of component-based product line engineering. It is based on the definition of variable architectural elements; and how they are derived from a given feature model and the interactions among features, providing a prescribed way to provide an explicit mapping between features and the variable architectural elements composing the product line architecture. For this purpose, we built a product line metamodel, connecting the concepts of software architecture with the concepts of variability and feature modeling. In this metamodel, a set of variability concepts are defined and used in the second part of the approach, which is the product line design method. This method is a component-based method, and is composed by two workflows: (i) a product line architecture design workflow to construct a product line architecture and a decision model for the product line; and (ii) an application architecture design workflow to generate a software architecture for a given product of the product line, using the product line architecture and the decision model as input. These workflows can integrate the design phases of a product line engineering process and an application engineering process, respectively.

The main contribution of this work is to provide a prescribed way to map a given set of features and interactions among features of a product line into the architectural elements of a product line architecture, enabling the decision resolution process to be automatized, and facilitating the maintenance of the product line, by adding or changing features. The proposed product line design method can be used as an integrated solution inside a component-based product line engineering process, assuming that the product line requirements and analysis phases were executed first, as a pre-requisite. Furthermore, the proposed solution proved to be prescriptive enough to be used in a real software product line context, the GoPhone SPL. This work is also innovative in regard to the use of feature interactions as an input to construct the product line architecture.

The main limitation of this approach regards to the architectural logical view. In spite of including an activity to select the logical view of the architecture, the method mainly focus on information systems using layered architectural style.

This work is an evolution of a previous work for managing architectural variabilities in software product lines [14]. Related to this previous work, some enhancements were introduced into the method, as feature interaction analysis and logical architectural design and architectural configuration. Future work includes, but is not restricted to the following issues: (i) detail the feature interaction analysis phase and provide guidelines on how feature interactions affects the product line architecture being constructed; (ii) provide guidelines

on using different architectural configurations; (iii) extend this approach to support product line evolution by adding or changing features; and (iv) provide tool support to automatize the instantiation of products, extending an already existing integrated environment for component-based development [19].

References

[1] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[2] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[3] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[4] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice, John Wiley & Sons*, 10(2):143–169, 2005.

[6] Edson A. de Oliveira Junior, Itana M. S. Gimenes, Elisa H. M. Huzita, and Jos Carlos Maldonado. A variability management process for software product lines. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2005.

[7] Stefan Ferber, Jrgen Haag, and Juha Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Software Product Lines : Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22*, volume Volume 2379 of *Lecture Notes in Computer Sciences*, pages 37–60. Springer Berlin / Heidelberg, 2002.

[8] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[9] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, Carnegie Mellon University – Software Engineering Institute, 1990.

[11] Rick Kazman and Len Bass. Toward deriving software architectures from quality attributes. Technical report, Software Engineering Institute, 1994.

[12] Kwanwoo Lee and Kyo C. Kang. Feature dependency analysis for product line component design. In *International Conference on Software Reuse (ICSR)*, 2004.

[13] Yuqin Lee, Chuanyao Yang, Chongxiang Zhu, and Wenyun Zhao. An approach to managing feature dependencies for product releasing in software product lines. In *ICSR*, pages 127–141, 2006.

[14] Ana Elisa C. Lobo, Patrick H. da S. Brito, and Ceclia M. F. Rubira. Managing Variabilities in Component-Based Software Product Lines. In *VI Workshop de Desenvolvimento Baseado em Componentes (WDBC)*, 2006.

[15] Dirk Muthig, Isabel John, Michalis Anastasopoulos, Thomas Forster, Jrg Drr, and Klaus Schmids. GoPhone - A Software Product Line in the Mobile Phone Domain. Technical Report IESE-Report No. 025.04/E, Version 1.0, Fraunhofer IESE (Institut Experimentelles Software Engineering), 2004.

[16] pure-systems GmbH. pure::variants - Variant Management. http://www.pure-systems.com/3.0.html/, 2006.

[17] Periklis Sochos, Matthias Riebisch, and Ilka Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS06)*, 2006.

[18] Steffen Thiel and Andreas Hein. Systematic integration of variability into product line architecture design. In *Software Product Lines Conference (SPLC)*, 2002.

[19] Rodrigo Teruo Tomita. Bellatrix: An Environment to Support Architecture-Centric Component-Based Development (in portuguese). Master's thesis, State University of Campinas, 2006.