Deriving Software Product Line's Architectural Requirements from Use Cases: an Experimental Approach *

Alexandre Bragança[1] and Ricardo J. Machado[2]

[1] Dep. Eng. Informática, ISEP, IPP, Porto, Portugal,
alex@dei.isep.ipp.pt
[2] Dep. Sistemas de Informação, Universidade do Minho,
Guimarães, Portugal,
rmac@dsi.uminho.pt

Abstract

One of the most important artifacts of a product line is the product line architecture. In this paper we present an approach for deriving a product line's architecture from the requirements of the product line. This approach is based on a transformational technique that has been developed and applied to obtain system architectures from requirements specified as UML use cases. In this paper we evaluate if such a technique can be applied to product lines and, if so, what adaptations are required. For presentation purposes we use the public available IESE report of the *GoPhone* product line that uses the UML modeling language.

1. Introduction

One of the most important artifacts of a product line is the product line architecture. A product line architecture is the basis for the derivation of the architectures of the members of the product line and also of the development of reusable product line components. As such, the product line architecture must encompass all the actual members of the product line as well as future members. This makes it a crucial artifact of the product line engineering process.

As for single systems development, the reference architecture for a product line is basically obtained from requirements. UML use cases are a widely adopted technique for functional requirements modeling. They are used with this perspective in single system development and also in product line approaches [1].

* This work has been supported by project *STACOS* (POSI/CHS/48875/2002).

In a product line approach requirements result from domain analysis. The domain analysis phase of product line engineering may involve several specific activities, besides functional requirements modeling, such as product line scoping and product portfolio definition.

The scoping activity aims at defining the products that the product line may include. In order to do so it is necessary to identify what is the domain of the product line and what are external and sub-domains. The result is usually a diagram representing the relations between domains.

The product portfolio aims at identifying the exact members of the product line, its characteristics and the timing for its development. To differentiate between products of the product line it is necessary to identify its features. Some features are common to several members of the product line while others are not. Feature diagrams are usually adopted for this purpose [2].

The two major techniques for dealing with requirements in a product line approach are use cases and feature models. They can be used together: the use case model is user oriented while the feature model is *reuser* oriented [3]. In this way, use cases focuses on requirements elicitation (what functionality should by provided by the product line), while features address better the functionality that can be *composed* for the members of the domain.

Regarding the reference architecture of a product line, use case models are the driving force that guides its development. Nevertheless, there are not documented processes in the product line area to help in the transition from use case requirements to high-level reference architectures. For instance, RSEB[4] (Reuse-driven Software Engineering Business) proposes that each use case gives origin to three kinds of objects, following the boundary-control-data pattern. But this is still just the starting point of the process. Other methods, like PuLSE [5], simply provide a framework for guiding the design and evaluation of the product line architecture.

In this context, we find that the derivation of a high-level architecture from the requirements of a product line is still a topic of the product line engineering process that needs further research. In this paper we address this problem. Our approach is based on a proven technique that has been used for the derivation of single system architectures from requirements modeled as UML use cases [6]. The 4SRS (4-Step Rule Set) technique applies transformational steps in order to derive a high-level architecture (system-level object model) from the requirements of a system. In order to use this technique in the product line context, adaptations are needed. For instance, the technique has to address the variability concept that is essential to product lines. In order to best evaluate our approach we use, along the paper, the publicly available IESE *GoPhone* product line technical report [7]. This technical report presents a mobile phone product

line engineered using PuLSE and KobrA. KobrA is an object-oriented customization of the PuLSE method [8].
The remainder of this paper is structured as follows. Section 2 discusses product line requirements modeling based on use cases. Section 3 describes the application of the 4SRS to derive an architecture for the *GoPhone* product line. It also discusses the modifications required to adapt 4SRS to product lines. The 4SRS resulting logical architecture is presented in Section 4 and a comparison is made with the architecture of the original *GoPhone* from the IESE report. This Section also addresses the instantiation process of architectures for members of the product line and the role of feature diagrams. Section 5 concludes the paper.

2. Requirements Modeling

Functional requirements of product lines can be modeled by use cases. Use case modeling in a product line must capture the requirements for all the possible members of the product line. As such, when adopting use cases to model the requirements of a product line, the major issue is the representation of variability. This means that each use case can vary, depending on the functional requirements of the members of the product line.

Variability is usually modeled using the concept of *variation points*. These variation points identify locations where variation will occur. In use cases, variation points can be expressed in different ways: includes relationship, extension points and use case parameters. To our knowledge, extension points are the more common way of expressing variability in use cases.

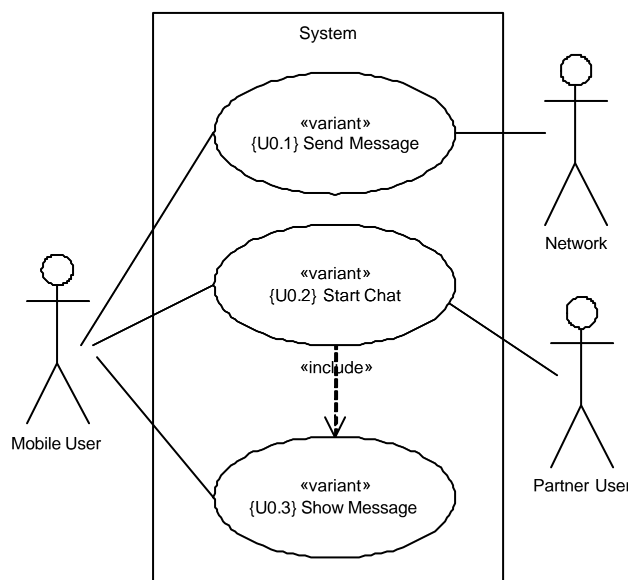

Fig. 1. Use case diagram depicting the main functionality of the messaging domain (Based on the IESE's GoPhone Technical Report [7])

Variability can also be modeled in use case diagrams by using stereotypes to mark use cases. For instance, Gomaa proposes three stereotypes to classify use cases regarding variability: «mandatory», «optional» and «alternative» [1].

Send Message

1. The user chooses the menu-item to send a message.
2. The user chooses the menu-item to start a new message.
3. Are there various message types?
<OPT> The system asks the user which kind of message he wants to send (Go Phone S, M, L, XL, Elegance, Com, Smart)
4. The system switches to a text editor.
5. The user enters the text message.
6. Is T9 supported?
<ALT 1> If T9 is activated, the system compares the entered word with the dictionary. (Go Phone XS, S, M, L, XL, Elegance)
7. Which kind of objects can be inserted into a message?
<ALT 1> The user can insert a picture into the message (Go Phone S, M, L, XL)
<ALT 2> The user can insert a picture or a drafted text-element into the message. (Go Phone Elegance, Com, Smart)
<ALT 3> Ø (Go Phone XS)
8. Which kind of objects can be attached to a message?
<ALT 1> The user can attach files, business cards, calendar entries or sounds to the message. (Go Phone Smart)
<ALT 2> The user can attach business cards or calendar entries to the message.(Go Phone S, M, L, XL, Elegance, Com)
<ALT 3> Ø (Go Phone XS)
9. The user chooses the menu-item to send the message.
10. The system asks the user for a recipient.
11. Which kind of message will be sent?
<ALT 1> The user types the phone number or chooses the recipient from the addressbook.(Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> In case of a basic or extended SMS, the user types the phone number or chooses the recipient from the addressbook. In case of an email, the user types the email-address or chooses the recipient from the addressbook. (Go Phone Com, Smart)
12. The system connects to the network and sends the message, then the system waits for an acknowledgement.
13. The network sends an acknowledgement to the system.
14. The system shows an acknowledgement to the user that the message was successfully sent.
15. Is a sent message directly saved in the sent-message folder?
<ALT 1> The system asks the user if the message should be saved. If it should be saved, the system saves the message in the 'sent-message' folder (Go Phone XS, S, M, L, XL, Elegance)
<ALT 2> The system saves the message in the 'sent-message' folder.
(Go Phone Com, Smart)
16. The system switches to the main menu.

Fig. 2. Description of the use case *Send Message* (Based on the IESE's GoPhone Technical Report [7])

In GoPhone, a variant use case has the stereotype «variant». A variant use case is a use case which functionality can vary between elements of the product line. Figure 1 shows the use case for the messaging domain of the GoPhone product

line. From the model it is possible to observe that *send message* is a variant use case of the product line. Further details regarding the use case variability are specified textually, in the use case description. Figure 2 is an extract from the textual documentation of the *send message* use case in the GoPhone report [7].

The *send message* description shows all the variation points of the use case. Variation points are identified by *OPT* or *ALT* tags. This approach explicitly points out all variation points of the use case but has disadvantages. For instance, if the use case is long, it may become very difficult to recognize a possible scenario for a member of the product line. Even further, this textual description is not adequate when the aim is the automation of tasks or the adoption of tools for dealing with variability.

In order to ease the automation of transforming the requirements of a product line into its high-level architecture (i.e., apply the 4SRS technique) we propose the explicit representation of the variation points in the use case model. In order to do so, a careful analysis of the initial use cases must be done.

The initial use cases, that are used to communicate the system functionalities with the stakeholders, must be transformed in order to express explicitly the functional variations of the product line. This activity can be done without the intervention of the users of the system. The main idea is to extract *include* and *extend* relationships from the textual description of the use cases. The *include* relationships will result from functional decomposition and will allow the discovery of functional commonalities among use cases. The *extend* relationships will basically result from extracting alternative and optional functionality from the use cases.

We like to view these activities as the construction of a three dimensional space representing the functionality of the product line: commonality, detail and variability. For instance, for each use case, we can go deeper (y axis) and broader (x axis) by adding detail as we do functional decomposition and find commonality. In a third dimension (z axis) we can express variability. This approach simplifies use case diagrams when requirements are extensive and complex because, for a given use case, one can choose to view only one perspective from the three dimensional space. In our approach we focus only on product line variability, i.e., functionality that can vary according to product line members. Variability that is common to all members of the product line can also be represented in the use case diagrams. But this can clutter the diagrams. We also advocate that this kind of variability can be better expressed in other types of diagrams like, for instance, activity diagrams. In this paper we will only address product line variability.

Next we briefly present how to construct the three dimensional space of use cases.

Functional decomposition

The initial use cases of the product line should be developed following, for instance, the process described by Alistair Cockburn [9]. This should result in use cases with a main scenario description similar to the one presented in Figure 2. These use cases should be at a medium level of detail, also know as *user level*. Based on these initial use cases, an analysis should be made with the goal of factor out fragments that have high degrees of commonality between them. For instance, regarding the messaging domain of the GoPhone product line we have found three of such fragments that have become the use cases `{U0.1.1} Choose Recipient` (steps 10 and 11 of Figure 2), `{U0.1.2} Compose Message` (steps 3 to 8 of Figure 2) and `{U0.1.3} Send Message to Network` (steps 12 to 14 of Figure 2). These use cases are common to the initial use cases `{U0.1} Send Message` and `{U0.2} Start Chat`. According to the 4SRS technique, each use case name is prefixed, within curly brackets, with a 'U' followed by period separated numbers denoting the level of the use case.

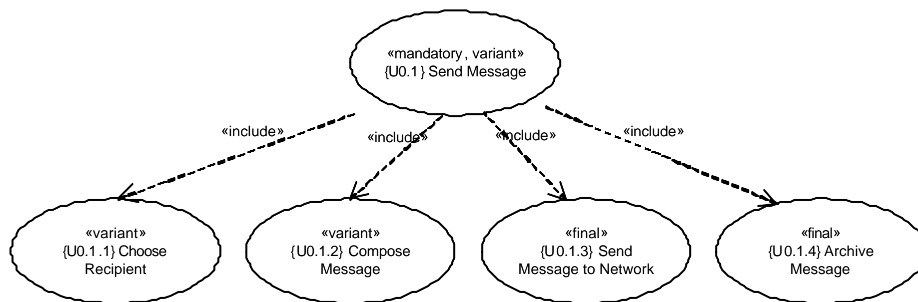

Fig. 3. Decomposition of use case `{U0.1} Send Message`

We adopt Gomaa's notation [1] for classifying use cases regarding their inclusion in the product line. As such, use cases can be marked with the stereotypes *mandatory*, *optional* or *alternative*. A mandatory use case is a use case that has to be included in all members of the product line. Optional and alternative use cases are only included in the members of the product line according to an inclusion condition. Alternative use cases must be in a group where usually one of the use cases is the default. This classification provides a very good foundation for viewing and analyzing the use case model according to the features of possible members of the product line.

When decomposing use cases, it is best to express the conditions regarding product line membership in the relationships, not the use cases. The reason is that these use cases can be included in several parent use cases, and the inclusion can vary depending on the parent. In Figure 3, `{U0.1} Send Message` has the stereotype *mandatory*, stating that this user level use case is to be included in all members of the product line. All the included relationships are mandatory, meaning that the use case `{U0.1} Send Message` requires all of the included

use cases. Regarding decomposability, the *final* stereotype indicates that the use case is not decomposable any further. We also propose the stereotype *abstract*, to mark use cases which have all their functionality realized by others use cases, as a result of the decomposition. Since the default stereotype for the include relationship is *mandatory*, the diagram of Figure 3 does not show this keyword near the relationships. To be noted that non-mandatory functionality regarding `{U0.1} Send Message` should be left to the variability perspective.

Variability externalization

The presented stereotypes do not provide hints regarding the variability of the use cases. So, in order to also express this information in the use case model we use the *variant* stereotype. When this stereotype appears on a use case it means that the use case has variability at the level of the product line. For instance, in Figure 3, use case `{U0.1.2} Compose Message` has the stereotype variant. This means that, at the product line level, this use case is variable. According to our three dimensional approach, Figure 4 presents `{U0.1.2} Compose Message` in the variability perspective (z-axis). The extension points of the use case are visible and also are the conditions of inclusion of the extending use cases, according to the UML 2.0 notation. The information required to construct these perspectives can be easily extracted from use case textual descriptions. For instance, all the information required for Figure 4 can be extracted from Figure 2.

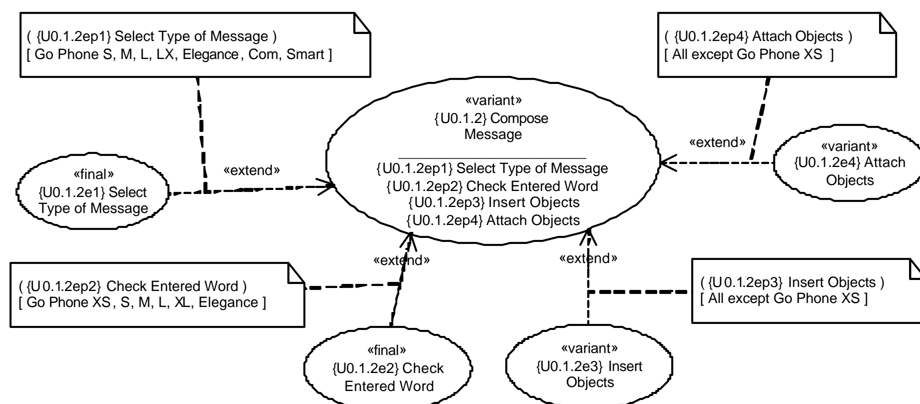

Fig. 4. Variability perspective of use case `{U0.1.2} Compose Message`

3. Architecture Derivation

This Section presents the application of the 4SRS technique to the GoPhone product line use case models. We basically present a description of the transformational steps with some examples to better explain the involved transformations.

3.1 Step 1 – Object Creation

In this step, each use case originates three objects. This operation follows the same approach as RSEB, which proposes the creation of three objects for each use case: an *interface* object, a *control* object and a *data* object. For instance, in the example of Figure 3, the use case `{U0.1} Send Message` originates three objects: `{O0.1.i}`, `{O0.1.c}` and `{O0.1.d}`. This is an automatic step, and also a blind one, since each and every non-abstract use case originates three objects. Each object is named according to the corresponding use case with a suffix that identifies the type of object.

Regarding the original technique, the adaptation required for dealing with product lines is the need to detail the use case diagrams with all the extension points. For instance, in the GoPhone case, this detail is never exposed in the use case model. The variability points are only described within the use case main scenario.

3.2 Step 2 – Object Elimination

This step of 4SRS is aimed at eliminating the unnecessary objects that resulted from the previous step. After this step, the object model should have only the objects that are functionally required, according to the requirements of the product line. The original 4SRS technique also states that "this step also supports the elimination of redundancy in the user requirements elicitation, as well as the discovering of missing requirements".

This is a major step of 4SRS and is comprised of several micro-steps.

Micro-step 2i: use case classification
In this micro-step, each use case is classified according to the *interface-control-data* heuristic that was used to automatically generate the objects in the previous step. The idea is that the classification of a use case can be a hint to eliminate unnecessary objects. Use cases are then classified according to one of the possibilities: "Ø", "i", "c", "d", "i-c", "i-d", "c-d", "i-c-d". Each letter is associated with one of the *interface-control-data* possibilities: "i"-*interface*, "c"-*control* and "d"-*data*. For instance, `{U0.1.4} Archive Message` is classified as "d", while `{U0.1.2e2} Check Entered Word` is classified as being "c-d".

Micro-step 2ii: local elimination
This micro-step regards the possible elimination of objects following the classification of the use cases in the previous step. To assist in this task, the description of the use cases should be used. For instance, the use case `{U0.1.2e2} Check Entered Word`, that was classified as being of type "c-d", is described in the GoPhone report as "If T9 is activated, the system compares the entered word with the dictionary". The value of this use case is

based on the T9 functionality for validating and suggesting words. As such, the control and data facets are much more important than the interface. According to this, the object {O0.1.2e2.i} is removed from the object model.

Micro-step 2iii: object naming
This micro-step aim is to give proper names to objects that were not removed in the previous micro-step. Names can be derived from the base use case name, the description of the use case and also the classification of the object. For instance, object {O0.1.2e2.d} is named as Word Repository.

Micro-step 2iv: object description
All the existing objects should have a description. According to 4SRS, this description should be based on the use case description from which they resulted. Next we present an example of such a description.

{O0.1.2e2.c} Word Validator: *This object checks words as they are entered by the user. This functionality is typical of phones that have the "T9" feature. For checking and memorization of words, the object uses object {O0.1.2e2.d} Word Repository.*

Micro-step 2v: object representation
The aim of this micro-step is to globally validate the model. For instance, redundancy can be discovered and removed. Basically, this step performs a semantic validation of the object model and also of the use case model. For instance, objects {O0.1.2e3e2.d} Picture Insertion, {O0.1.2e3e1.d} Draft Text Insertion, {O0.1.2e4e2.d} File Attach and {O0.3e3e1.d} File View and Save all represent the functionality of a repository of files. As such, we maintain only {O0.1.2e4e2.d} File Attach, since the semantic of this object includes the functionality of the other three objects.

Micro-step 2vi: global elimination
This is an "automatic" micro-step, since it is based on the results of the previous one. This step eliminates all the objects that were considered redundant in the previous step. For instance, resulting from the last micro-step, the objects {O0.1.2e3e2.d}, {O0.1.2e3e1.d} and {O0.3e3e1.d} are removed, since its functionality can be provided by the object {O0.1.2e4e2.d} File Attach. The result of this micro-step is a minimum number of objects that represent the product line functional requirements.

Micro-step 2vii: object renaming
The aim of this micro-step is to rename the objects that were not removed in the previous micro-step and that represent other objects. The documentation of such objects must also be updated. For instance, the {O0.1.2e4e2.d} File

`Attach` object is renamed `{O0.1.2e4e2.d} File Repository` to proper represent its functionality, taking into account all the previous objects it represents.

3.3 Step 3 – Object Packaging & Aggregation

In this step, objects that make sense to be treated in a unified way can be placed in the same package. Aggregation can also be applied if there is a strong relationship between objects. This is usually the case of legacy objects in a sub-system. In the GoPhone product line this is not the case.

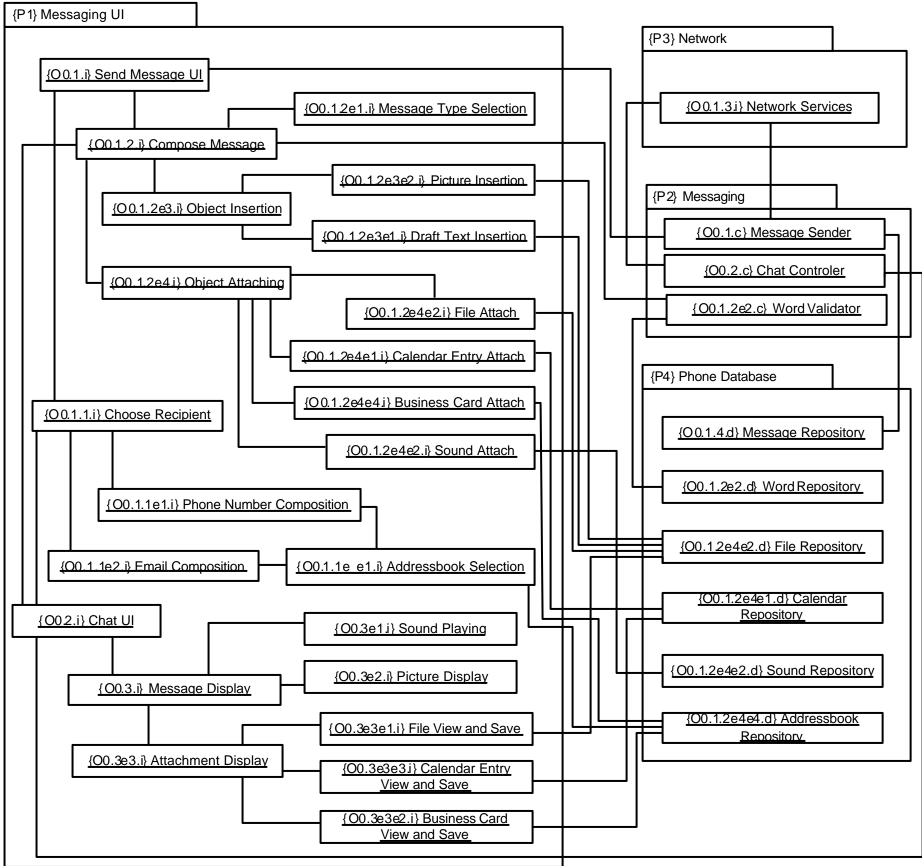

Fig. 5. Object model of the messaging domain

Since we are dealing only with the messaging domain of the product line, the packaging of objects follows this fact. As such, objects representing the user interface of the messaging domain are packaged in `{P1} Messaging UI` and objects representing messaging controlling and behavior are packaged in `{P2} Messaging`. Objects which major functionality is data persistence are included

in {P4} Phone Database. We call this package *phone database* and not *messaging database* because it archives data regarding not only messages but other phone concepts like, contacts or files. {P3} Network is a package that includes objects with functionality regarding the mobile network, i.e., they represent the interface between the mobile phone and the network.

3.4 Step 4 – Object Association

This step introduces associations between objects that can be obtained from micro-step 2i. Also the relations between use cases can be used to generate associations between objects.

This is the last step in the 4SRS technique. Figure 5 presents the resulting object model for the messaging domain, including the packages. This object model, which resulted from the application of the 4SRS technique, is a system level object model. It provides high-level guidelines for the next phases of the development process. As such, it provides the basis for the requirements of a logical architecture that will support the following development phases. As it is possible to observe in Figure 5, the object model that result from the 4SRS technique includes all the functionality described in the source use cases. It is even possible to expose some hints regarding the product line variability, because, for instance, objects with an 'e' in their identification resulted from extending use cases. In the next Section we explore some issues regarding the logical architecture of a product line, namely variability representation and product member instantiation.

4. Logical Architecture

The major aim of a logical architecture is to serve as the basis for the design of a system. As such, it encompasses the description of the logical components of the system and also the interactions between them [10]. As presented in the previous Sections, the object model that results from 4SRS contains the components (objects) and interactions between them (object associations). As such, the object model that results from the 4SRS technique can be of great value for a system architect, because it clearly provides 'suggestions' for the logical components of a system and the interactions between them. This is very different from the usual gap that exists between requirements and the initial architecture for a system. This gap can be very 'dangerous' when the problem domain is new and there is not much knowledge in the solution space of the domain. In these cases it can be very difficult to design the system or even apply design patterns.

In the GoPhone technical report, the product line architectural design is based on the KobrA method and also on two patterns: the *mediator pattern* and the *state pattern*. The objective of the mediator pattern is to achieve changeability and extensibility of the components and, as such, achieve flexibility in the product

line. The justification for the state pattern is that it enables handling the small displays of mobile phones. These two patterns result from non-functional requirements: flexibility and state management. They impose some guidelines in the architecture but they do not provide information regarding the functional components of the architecture. This is what we propose to achieve with the adoption of the 4SRS technique: a semi-automatic technique to obtain the product line's architecture functional requirements. The object model presented in Figure 5, which resulted from applying the 4SRS technique, depicts a partial view of such requirements for the GoPhone system. With such a model it is possible to design the system by applying well-known patterns, such as the mediator and the state pattern (such as in the GoPhone report). The difference from the GoPhone report is that, with our approach, we know which logical functional components are necessary to incorporate in the design. In this case, our logical architecture for the GoPhone product line is very similar with the one from the original report, the major difference being the fact that in our process we did not adopt KobrA.

The 4SRS technique was originally designed for obtaining the logical architecture of single systems. For this reason it does not deal explicitly with variability. As we saw, the main resulting artifact of the 4SRS technique is the object model. In our experimental approach to adapt 4SRS for product lines we have already proposed the need to externalize variability in the use case model. Regarding the logical architecture we also propose that other views of the system are needed to properly address product line development requirements. For instance, a class model may be more appropriate to express variability at the architectural level. Also, activity models are more appropriate to express fine grained variability. As such, we propose a multiple model approach for 4SRS. A similar approach can be also find in [11].

This multiple model approach is also more suited to deal with product line member instantiation. Product line member instantiation is based on the selection of features required for the member being instantiated. As mentioned in Section 1, the usual approach is to build a feature diagram to guide this instantiation. The construction of a feature diagram can be done in parallel with the use case diagrams. In our approach, feature diagrams correspond to choices in the variability perspective (z-axis), when navigating through the use case model. A functional feature is basically realized by a use case. Extending use cases become optional or alternative features. Figure 6 presents a feature diagram for *send message*. The Figure also presents a possible example of the selection of features for a product line member, by showing them in gray.

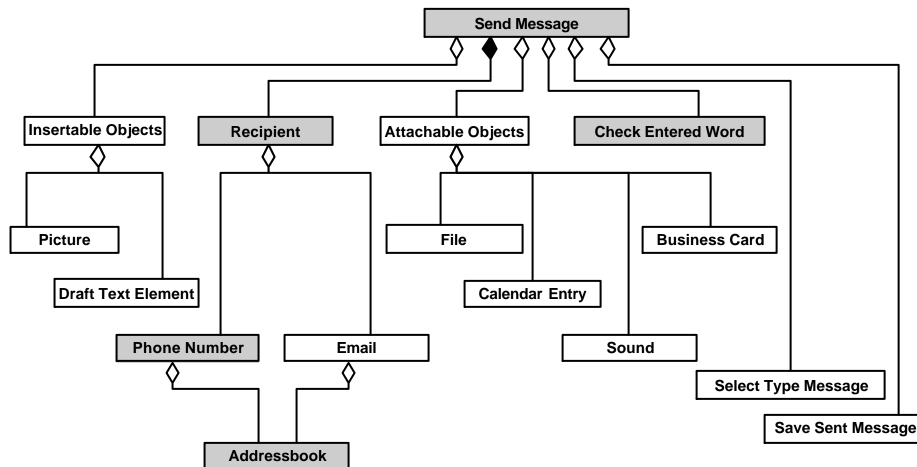

Fig. 6. Feature diagram for *Send Message* (Based on the notation proposed in [1])

Figure 7 presents an excerpt of a possible class diagram depicting the *send message* feature according to the feature selections of Figure 6. The class model should be constructed after the object model. The major goal of the class model, at this logical architectural level, is to be the first approximation to a meta-level structural model of the product line architecture. In the process of constructing the class model it is possible and even common that functionalities provided by several objects become realized by a single class or a hierarchy of classes.

Similar to the object diagram, the class diagram at this logical architectural level is used to represent the product line at a component level of abstraction. For the moment, we are not adopting component diagrams because they are best suited for modeling the system at a lower level of abstraction, particularly at the physical level.

The class model also provides a way to explicitly represent the product line variability. So, the construction of the class model is also based on the use case model and feature model. The description of the process for the construction of the class model is out of the scope of this paper. We intend to address it in our future work.

As it is possible to observe in Figure 7, in this experimental approach we have adopted outgoing and incoming interfaces to model extension points. This seams to be an appropriate choice at this logical component level. This option does not compromise later design decisions of how to realize the extension points. In fact, other authors have proposed comprehensive feature variability realization techniques at the design level that are based on interfaces [12].

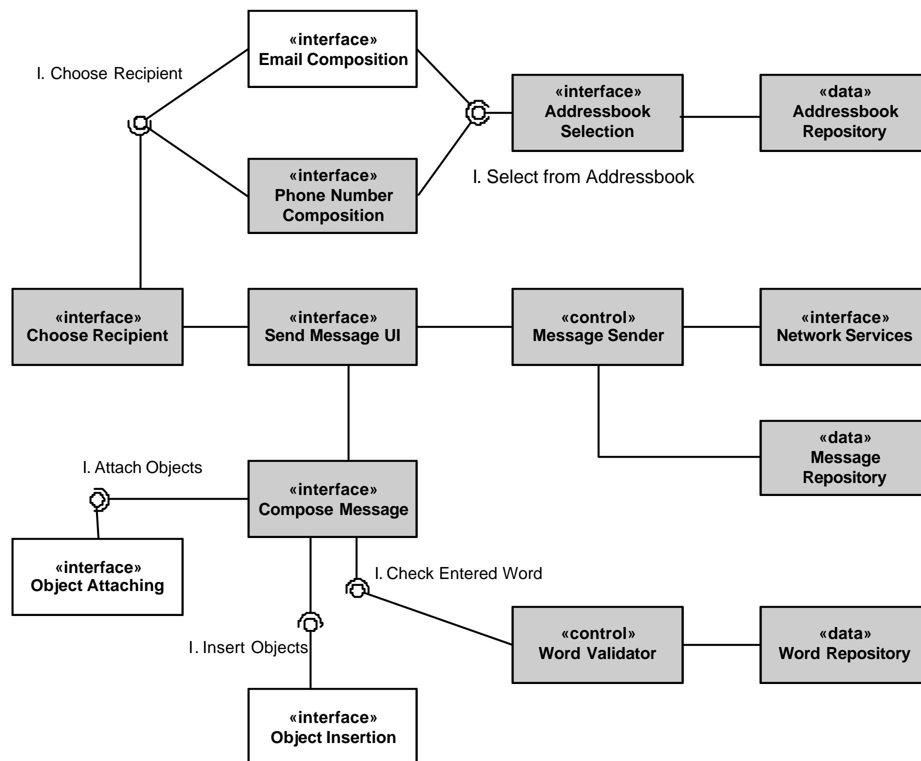

Fig. 7. Excerpt of class diagram for *Send Message*

Since in our approach there is a very direct mapping between the use case model and the feature model and because it is easy to keep trace links from the class model to the object model and ultimately to the use case model, it is possible to derive the architectural requirements for a product line member based on its features.

5. Conclusions

In this paper we have explored an approach for deriving a software product line logical architecture from its requirements, by adopting and adapting a transformational technique. We have focused the discussion in the transformational technique for obtaining an object model from the use case model. This is only a part of a multi-view and multi-model process approach for product line development. We intend to present and discuss more aspects of this process in our future work.

We have found the results of this experimental approach very promising. Nonetheless, several questions still remain open and require further validation. For instance, the tree dimensional view of the use case model seems to be a

requirement for dealing with very complex product lines. But we need to validate this with more case experiences. This will also provide a context to further explore the technique for feature diagram construction based on the use case variability perspective.

Another point open to further research regards variability representation. For the moment our approach only deals with component level variability. It seams, even in the GoPhone case, that more fine grained representation for variability is needed. This is true for use cases, object and class diagrams. In our future work we intend to approach this problem mainly by using activity diagrams in the context of use cases and also explore aspect oriented approaches as a way to deal with operation level variability. OCL seams also a promising approach to express variability in the model in a more formal way.

References

[1] Gomaa, H., *Designing Software Product Lines with UML*. Addison Wesley. 2005.

[2] Kang, K.C., J. Lee, and P. Donohoe, *Feature-Oriented Product Line Engineering.* IEEE Software, (July/August 2002). 2002.

[3] Griss, M.L., J. Favaro, and M. d'Alessandro. *Integrating Feature Modeling with the RSEB*. in *Fifth International Conference on Software Reuse.* Victoria, Canada. IEEE Computer Society Press. 1998.

[4] Jacobson, I., M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman. 1997.

[5] Anastasopoulos, M., J. Bayer, O. Flege, and C. Gacek. *A Process for Product Line Architecture Creation and Evaluation*. IESE. Technical report: 038.00/E. 2000.

[6] Machado, R.J., J.M. Fernandes, P. Monteiro, and H. Rodrigues. *On the Transformation of UML Models for Service-Oriented Software.* in *ECBS International Conference and Workshop on the Engineering of Computer Based Systems*. Greenbelt, Maryland. 2005.

[7] Muthig, D., I. John, M. Anastasopoulos, T. Forster, J. Dorr, and K. Schmid. *GoPhone - A Software Product Line in the Mobile Phone Domain*. IESE. Technical report: 025.04/E. 2004.

[8] Bayer, J., D. Muthig, and B. Gopfert. *The Library Systems Product Line - A KobrA Case Study*. IESE. Technical report: 024.01/E. 2001.

[9] Cockburn, A., *Writing Effective Use Cases*. Addison-Wesley. 2001.

[10] Garlan, D. and M. Shaw. *An Introduction to Software Architecture.* Carnegie Mellon University. Technical report: CMU-CS-94-166. 1994.

[11] Gomaa, H. and M.E. Shin. *A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines*. in *ICSR International Conference on Software Reuse.* Madrid. 2004.

[12] Lee, K. and K.C. Kang. *Feature Dependency Analysis for Product Line Component Design*. in *ICSR 2004 - International Conference on Software Reuse.* Madrid. 2004.