

Refinement of Software Product Line Architectures through Recursive Modeling Techniques

Sofia Azevedo¹, Ricardo J. Machado¹, Dirk Muthig², and Hugo Ribeiro³

¹ University of Minho, Portugal

{sofia.azevedo, rmac}@dsi.uminho.pt

² Lufthansa Systems, Germany

dirk.muthig@lhsystems.com

³ Primavera Business Software Solutions, Portugal

hugo.ribeiro@primaverabss.com

Abstract. Currently, modeling methods applicable to software product line architectures do not explicitly comprise refinement, which implies dealing with a lot of complexity during their application to a high number of requirements. This paper suggests the extension of a modeling method applicable to product line architectural modeling, the 4SRS (Four Step Rule Set), to support the refinement of product lines. We have used the GoPhone case study to illustrate the approach and the recursion capability of the method as a solution to the challenges of modeling product line architectures. The strength of our approach resides in its stepwise nature and in allowing the modeler to work at the user requirements level without delving into lower abstraction concerns.

Keywords: software product line architectural modeling, logical architecture, refinement, recursion.

1 Introduction

A logical architecture can be faced as a view of a system composed of a set of problem-specific abstractions supporting functional requirements and it is represented as objects or object classes [1]. In this paper, logical architectures will be represented as component diagrams. The software product line architecture, in the context of this contribute, is the design artifact representing the functionality-based structure of the product line, and embraces both the product line's reusable components and the foundation from which the architectures of the product line members can be isolated. The non-functional requirements are out of the scope of this paper, although they will have to be considered, most likely through the application of patterns, in moments following the product line logical architecture's design.

Product line architectures can be obtained from functional requirements expressed in use cases with variability exposure [2, 3]. Since use cases can be more or less detailed, architectural components can be *refined* according to the detail level of the use cases they originate from. But the relation between use case and component is not one-to-one. It may be the case that many components are affected by the refinement of a use case and many more than the original set.

The refinement of product line logical architectures may be due to two reasons: (1) the addition of detail to the product line logical architecture; and (2) the transition from a phase of user requirements to a phase of system requirements in the product line development.

Currently, the modeling approaches that support product line architecture design do not take refinement into consideration at the user requirements level, at the use cases level and in a stepwise way (see Section 2 for evidence to support this statement).

Not considering refinement when modeling a product line architecture using a specific product line modeling method, in our case, the 4SRS (Four Step Rule Set) [4], constitutes a problem of complexity in the modeling activity. The pertinence of refinement resides in large-scale contexts, even though we are going to demonstrate our approach to refinement with GoPhone [5], which is of small-scale. Our intention is to expose our contribution and the concepts we want the reader to understand in a straightforward way. In order to circumvent the scale problem, we propose to extend the 4SRS modeling method to support the refinement of product line architectures. The motivation for the extension of the 4SRS is not restricted to addressing a limitation or weakness of the method, rather it is based on an unsolved problem or need in the product line engineering as evidenced by the related work analysis we will present in the next section of the paper. We will address the recursive character of the 4SRS as solution for the refinement.

The Fraunhofer IESE's GoPhone case study presents a series of use cases for a part of a mobile phone product line, particularly concerning the interaction between the user and the mobile phone software. The use cases had to be extended with a variability mechanism for product line modeling. Actors and use cases can be variant if they are not supported by every product of the product family. Variant use cases can be alternative to other use cases or they can be optional (refer to Section *From Components to Use Cases* for more information on the different stereotypes that can be applied to the use cases). Figure 1a depicts the user level use cases concerning the messaging domain.

The remainder of this paper is structured as follows. Section 2 is targeted at describing the strengths of our approach when compared to other approaches. Section 3 describes the 4SRS general approach. Section 4 explains the filtering and collapsing techniques that we use in refinement contexts. Section 5 is dedicated to the technical details on the derivation of use cases from components. Section 6 affords the recursion of the 4SRS. Section 7 provides for some concluding remarks.

2 Related Work

There are other approaches to variability modeling besides the 4SRS, such as FODA, Kobra or FAST [6-8]. None of them explicitly approaches refinement through recursion incorporated in the modeling method itself.

Smaragdakis and Batory [9] mention refinement in their work on collaboration-based design of large-scale software applications, applicable to the design of product line architectures. In their approach, refinement is achieved through collaborations. In our approach, we use a method that considers user level requirements in the first

place, before dealing with artifacts that would reside in product line conception phases posterior to the product line architecture definition and refinement (artifacts like collaborations). This allows us to work at the level of the user requirements, without having to make finer grained design decisions in the beginning of the product line architecture's conception.

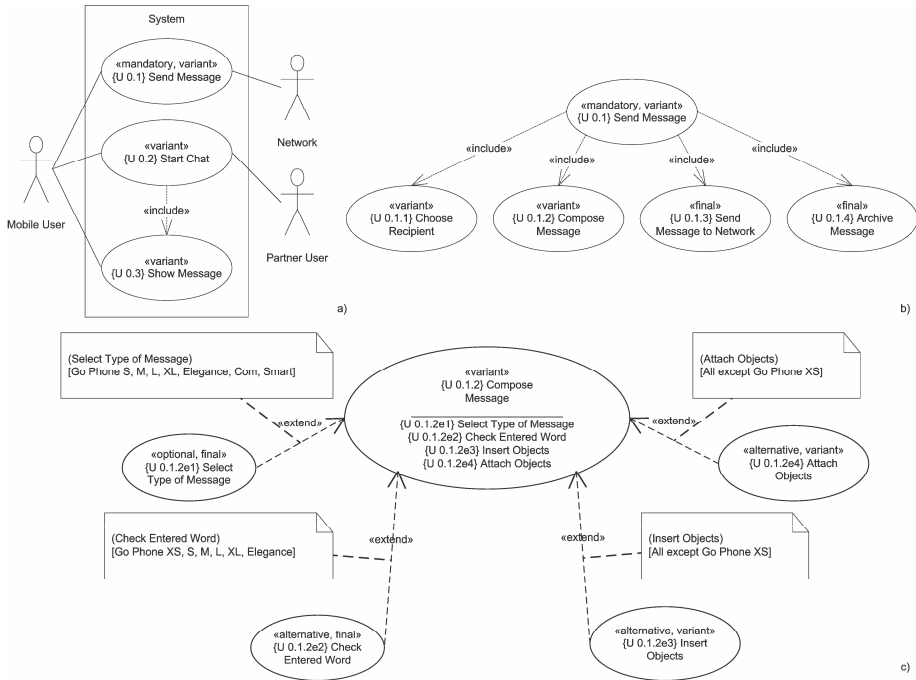


Fig. 1. Use case diagrams from the GoPhone case study

Working at the level of the user requirements is our strength when comparing our perception to the perception of refinement that also Greenfield and Short [10] and Egyed et al. [11] have.

Batory also created a model, the AHEAD model [12], for expressing the refinement of system representations as equations. Despite his approach being based on stepwise refinement, he worked at a code-oriented level. The work we are presenting in this paper allows refining, also in a stepwise manner, software models that shall be handled before code is handled during the software construction phase.

Gomaa [13] explored refinement in the context of feature modeling, where a feature can be a refinement of another. But in order to get to the features, use cases have to be modeled and mapped to features. Our approach eliminates the mapping activity. Eriksson et al. [14] have an understanding of refinement similar to the Gomaa's.

PuLSE comprises the refinement of partial product line architectures [15] (a partial product line architecture is a product line architecture that can be popped in the one

whose refinement originated it; product line architectures going to be refined are also partial product line architectures), but the refinement is not conducted in a stepwise manner. PuLSE's approach includes testing steps to assure that the architecture supports the requirements from which it was conceived. Ours doesn't include such kind of steps, yet. Despite that, our approach is stronger than PuLSE's in allowing the refinement in a stepwise, thus, guided mode.

The 4SRS is a method to allow the iterative and incremental model-based transformation of user level functional requirements in the shape of use cases into logical architectures in the shape of component diagrams. Originally, the 4SRS didn't support variability considerations in the modeling of software systems, which didn't allow for the method to support product line modeling. But after some extension, that was possible [2].

3 General Principals of the 4SRS

Shortly, the 4SRS is composed of the following steps: (1) *Component Creation*, whose goal is to create three kinds of components for each use case: an interface component, a control component and a data component; (2) *Component Elimination*, whose goal is to remove redundant requirements and to find missing requirements (this step is vital in order to validate the components blindly created in the previous step and it includes eliminating the components whose requirements are already represented by other components); (3) *Component Packaging and Aggregation*, whose goal is to semantically group components in packages; and (4) *Component Association*, whose goal is to introduce associations of components with each other into the component diagram.

In the past, the 4SRS has been extended to support tabular transformations in the execution of its steps, as well as some filtering and collapsing techniques to enable the refinement of logical architectures [4]. After that, the method has been extended to support product line logical architecture modeling, which added the notion of variability to it [2]. Our work in this paper is both the conjunction and the prosecution of these previous works on the 4SRS as it will allow for a systematic application of filtering and collapsing techniques for the refinement of product line logical architectures.

Software product line architectural modeling (see [1] for an example of a logical architecture representation through models) may be conducted with specific instruments tailored to explicitly expose the variability of the product line, like the 4SRS method with its extension to variability support. Architectural refinement is the approach the 4SRS takes to increment a primary logical architecture with detail (by primary we mean the baseline architecture, the architecture that is going to be detailed). Recursion, in the context of this paper, is the ability of the modeling method to call itself within the process of modeling the architecture. Recursion leads to various executions of the method, an initial architecture and various partial architectures. The number of partial architectures is equal to the number of executions minus one.

According to what was just stated, the method can be applied to product line architectural modeling. As depicted in Figure 2, the 4SRS may be applied recursively, in several *executions*. In the context of each one of those executions, various *iterations* can be performed. Although there is no stopping rule for iterating over the same use case diagram, it can be performed until the results obtained generate a logical architecture that does not benefit from additional iterations in terms of the

elimination of redundant requirements, the finding of missing requirements and the increasing of the cohesion of the logical architecture itself. In the case of refinement (by recursion), when one of the executions is considered to be finished by the modeler, the output of that execution's last iteration is going to be transformed into the input of the subsequent execution's first iteration. The task flow of the new execution is exactly the same as the task flow of the preceding one. Again, in the case of refinement (by recursion), the logical architectures the various executions produce over time are situated in lower levels of abstraction and cover less functionality than their preceding ones.

Roadmap of the Extension

We divided the extension of the 4SRS to product line architecture refinement into two parts: before the method execution and during the method execution. Before the recursive execution of the 4SRS, three steps must be performed: (1) the filtering and collapsing; (2) the derivation of use cases from components; and (3) the detailing of use cases. The following is the new sequence of steps for the 4SRS: (1) *Component Creation*; (2) *Component Elimination*; (3) *Component Packaging and Aggregation*; (4) *Component Association*; (4+1) *Filtering and Collapsing*; and (4+2) *From Components to Use Cases*. The first four steps are the original steps and the other ones are what we call the new *intermediate steps*, which are performed in between executions of the 4SRS.

We have extended step 4 of the 4SRS with rules for the association of components with each other, based on the inclusion and the extension of use cases, as well as on the categorization of components, according to the *interface-control-data* heuristic, performed when they are created in step 1.

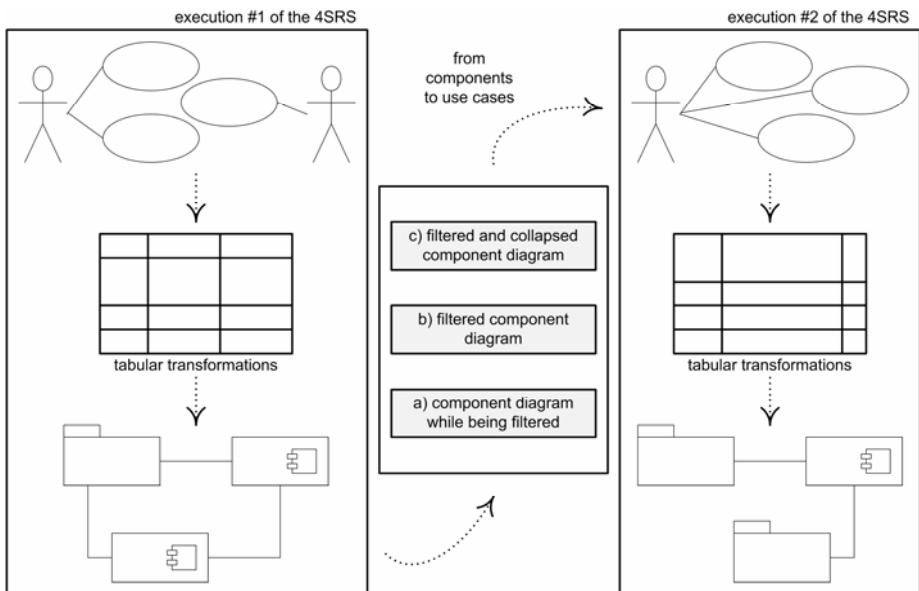


Fig. 2. Schematic representation of the recursive execution of the 4SRS

4 Filtering and Collapsing

We use techniques of filtering and collapsing in between executions of the 4SRS to achieve the refinement of the input product line logical architecture. The filtering is about allowing subsystem partitioning for further subsystem design and specification. The collapsing is about defining the border of the subsystem whose components are going to be refined. Later on, those components will be replaced, inside the limits of that same border, by others of lower abstraction level or higher detail level.

The filtering is what we, in the current work, call the *selection of components*. This selection is concerned with choosing the components that shall be the input for the subsequent recursive execution of the 4SRS. Earlier, the filtering activity of the 4SRS consisted of considering some components as the subsystem for refinement and discarding those that were not related to them [4]. Now, in this work, the same applies, but we have systematized the choice of the most adequate components for refinement. Components can be classified into *filtering types*: (type 1) the components to which we may apply refinement; (type 2) the components to which we cannot apply refinement; and (type 3) the components that we are actually going to refine. The components in Figure 3 that have a white background are of type 1, whereas the ones marked with a cross are of type 2 and the ones with a grey background are of type 3.

Components originated from use cases not marked with the stereotype «*variant*» (consider that this stereotype has been omitted from included use cases) have been classified as type 2 components. From the remaining components, the ones originated from decomposable use cases (those use cases not marked with the stereotype «*final*»; decomposable use cases cannot be either included use cases or extension points that cannot be decomposed any further) that are simultaneously *leaf* use cases have been classified as type 3 components. Despite type 2 components being originated from *leaf* use cases, they are neither longer *refinable* through inclusion relationships, nor through extension relationships at the use cases level. From the type 1 and the type 3 components, the type 1 components can be «*final*», therefore, not decomposable, or they can be not *leaf* use cases, which means that they are no longer extendable. Therefore, the reduction of the components to the selected ones is very much coupled with the decomposition extension capacity of the requirements they represent.

For instance, the component $\{C\ 0.1.1.1e1.i\}$ *Phone Number from Address Book Choice UI*, from Figure 3, originated from the use case $\{U\ 0.1.1.1e1\}$ *Choose Phone Number from Address Book*, in Figure 4, has been classified as of type 2, since that use case has not been tagged with the stereotype «*variant*», which makes of it a variant use case.

After the filtering is performed, the result obtained is the diagram in Figure 5. Only the type 3 components, along with the component which is associated with them (in this case it is only one) appear in the figure. The actor that relates to all the components is also represented in the diagram.

Earlier, the collapsing activity of the 4SRS was about hiding package details [4]. Currently, in this work, the same applies with no change. As we can see from Figure 6a, the collapsed diagram is an evolution of the filtered diagram in Figure 5, as it presents a subsystem in place of the once classified as type 3 components. As the components have been removed from the subsystem to refine, the associations

between the component *{C 0.1.2.i} Message Composition UI* and those components have been removed as well and replaced by two interfaces, one for each of the components. The new components that will be placed inside the borders of the subsystem will have to comply with those interfaces.

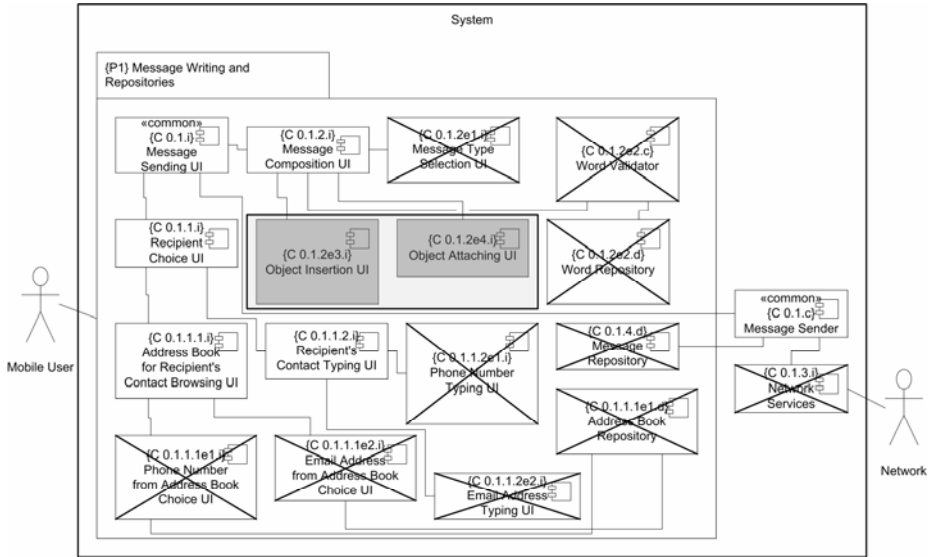


Fig. 3. Component diagram, which resulted from the first execution of the 4SRS, with variability support, to the *Send Message* use case of the GoPhone case study, while being filtered

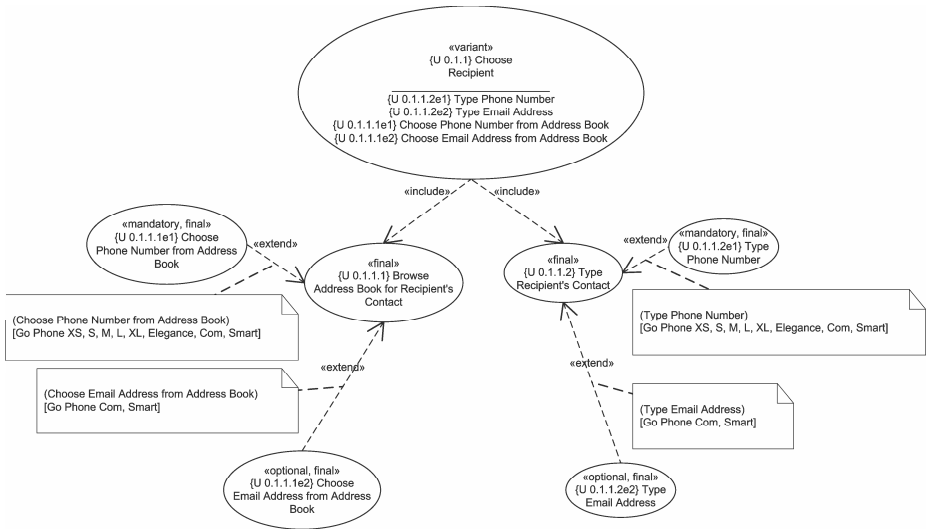


Fig. 4. Choose Recipient extension points

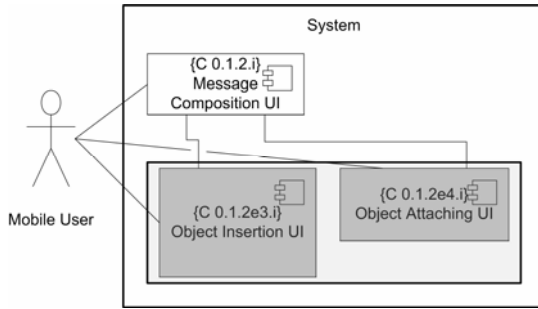


Fig. 5. Filtered component diagram with regards to the object insertion and object attaching functionalities of the *Send Message* use case from the GoPhone case study

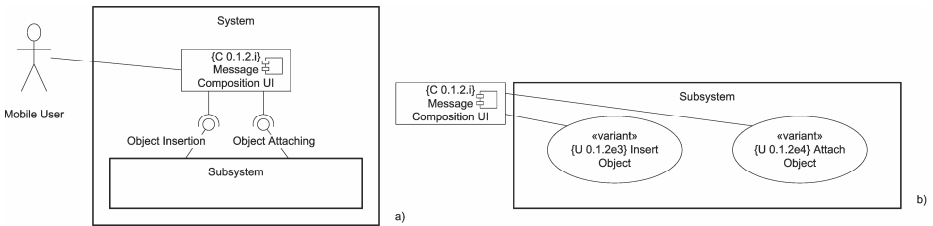


Fig. 6. a) Filtered and collapsed diagram for object insertion and attaching; b) Use case diagram for the first recursive execution of the 4SRS

5 From Components to Use Cases

This intermediate step is composed of two intermediate substeps. The goal of the first one, the *Deriving Use Cases from Components*, is to derive, from the component diagram, the use cases to hand out as input for the succeeding recursive execution of the 4SRS. This is an important procedure as it will define the use cases that will feed the refinement process. These use cases are an expression of the requirements that the architecture of the subsystem defined in the collapsed diagram must represent.

The components that survived the selection process were the *{C 0.1.2e3.i} Object Insertion UI* and the *{C 0.1.2e4.i} Object Attaching UI*. A use case diagram with the use cases derived from the components is depicted in Figure 6b. The use cases in the subsystem were defined taking into consideration the descriptions of the components that survived the selection process (those descriptions were elaborated during the execution of the prior step 2 from the 4SRS). It was concluded that there was no need to create new actors, based on those descriptions, so, the component which was associated with the subsystem in the collapsed diagram of Figure 6a is the only actor in the use case diagram of Figure 6b.

The goal of the second intermediate substep, the *Detailing Use Cases*, is to elicit, from the use cases in the diagram in Figure 6b, the inclusion and extension relationships that will further on steer the recursive execution of the 4SRS. The use cases must be detailed in order for them to be decomposed through inclusion relationships. After determining the included use cases, they are analyzed in terms of

extension points, which originate the extending use cases. Next, we have to materialize, by means of diagrams, the inclusion and extension relationships. The result is going to be a use case model with the use case diagram in Figure 6b and another use case diagram for each one of the use cases in it with its included and extending use cases.

Comparing the description of the use case *Send Message* in Figure 7 and the description of the use case *Insert Object* in Figure 8, we can see that the last one is the detailing of the three alternatives the step 7 comprised in the *Send Message*'s description. The step 7 of the *Send Message*'s description is the one that gave origin to the use case $\{U\ 0.1.2e3\}$ *Insert Object*.

-
1. The user chooses the menu item to send a message.
 2. The user chooses the menu item to start a new message.
 3. The system may ask the user which kind of message he wants to send.
 4. The system switches to a text editor.
 5. The user enters the text message.
 6. If T9 is supported and activated, the system compares the entered word with the dictionary.
 7. Which kind of objects can be inserted into a message?
 <ALT 1> The user can insert a picture into the message.
 <ALT 2> The user can insert a picture or a draft text into the message.
 <ALT 3> \emptyset
 8. The user may attach objects to a message.
 9. The user chooses the menu item to send the message.
 10. The system asks the user for a recipient.
 11. The user chooses the recipient.
 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. The system saves the message in the sent messages folder, directly or upon user request.
 16. The system switches to the main menu.
-

Fig. 7. *Send Message* use case textual description

-
1. Which kind of objects can be inserted into the message?
 <ALT 1> The user can insert a picture into the message
 - 1.1. The system shows the user a browsable directory of folders and/or picture files
 - 1.2. The user browses the directory and selects a picture from it
 - 1.3. The picture is inserted into the message and displayed to the user in the message area
 <ALT 2> The user can insert a picture OR a draft text into the message
 <ALT 2.1> The user chooses to insert a picture into the message
 - 1.1. The system shows the user a browsable directory of folders and/or picture files
 - 1.2. The user browses the directory and selects a picture from it
 - 1.3. The picture is inserted into the message and displayed to the user in the message area
 <ALT 2.2> The user chooses to insert a draft text into the message
 - 1.1. The system shows the user a list of draft texts
 - 1.2. The user browses the list and selects a draft text from it
 - 1.3. The draft text is inserted into the message and displayed to the user in the message area
 <ALT 3> The user cannot insert any object into the message
-

Fig. 8. Detailing of the use case *Insert Object*

For instance, after analyzing the first two alternatives of *Send Message*'s step 7, we see that they have a common part, the insertion of a picture, which can be performed in each one of the alternatives. Hence, we repeated the detailing of the picture insertion in the detailing of alternatives 1 and 2. We have also divided the detailing of alternative 2 into the detailing of the picture insertion and of the draft text insertion, since these are functionalities of alternative 2 separated by a logical operator (*OR*). We are excluding the third alternative as there is no functionality attached to it. When detailing the functionalities of picture insertion and of draft text insertion, a common verbal expression emerged: *display*. The display object functionality is extendable by the display picture and the display draft text functionalities and included in the insert object functionality. Figure 9 illustrates a diagram of the object insertion functionality.

The following stereotypes must be applied to the use cases, which are going to be the input for the recursive execution of the 4SRS, according to the following rules: (1) «*mandatory*», applicable to the use cases present in all product line members and to the inclusion relationships; (2) «*alternative*», applicable to the extending use cases; (3) «*optional*», applicable to the extending use cases as well, but only when the requirement may not exist or the use case may not be performed in the product line member; (4) «*final*», applicable to the included use cases and to the extension points that cannot be decomposed any further; and (5) «*variant*», applicable to the use cases which functionality can vary between the product line members.

6 The Recursive Execution of the 4SRS

The goal of the recursive execution of the 4SRS is to actually apply the 4SRS after one or more executions of this modeling method have occurred beforehand. This process is about applying the tabular transformations 4SRS is composed of, as already explained in [4], to the target use cases. The result will be a list of components that shall be associated with each other. The association of components with each other can be an automatic process, based on predefined rules directly related to the inclusion and the extension relationships established between the use cases the components originate from. Hence, it is possible to systematically generate associations between components in the component diagram from the relationships established between the use cases in the use case diagram, as suggested in [2]. The value of this generation lies in the possibility of verifying the consistency of the component diagram with the system's functional requirements, as well as of avoiding too many iterations to each one of the executions of the 4SRS. Taking figures 1c, 3 and 4 as an example, the component *{C 0.1.2.i} Message Composition UI*, originated from the use case *{U 0.1.2} Compose Message*, is associated with the component *{C 0.1.2e1.i} Message Type Selection UI*, originated from the use case *{U 0.1.2e1} Select Type of Message*. In the use case diagram, these two use cases are connected through an extension relationship. Consider the existence, for a single use case, of all the three corresponding components resulting from the 4SRS' categorization, *i*, *c* and *d*. They shall always be connected in this order: the *i* component to the *c* and the *c* to the *d*. The resulting artifact from the execution of the 4SRS with refinement to the *Send Message* functionality of the GoPhone case study is the component diagram in Figure 10. This diagram has, then, to be integrated into the logical architecture the preceding execution of the 4SRS originated.

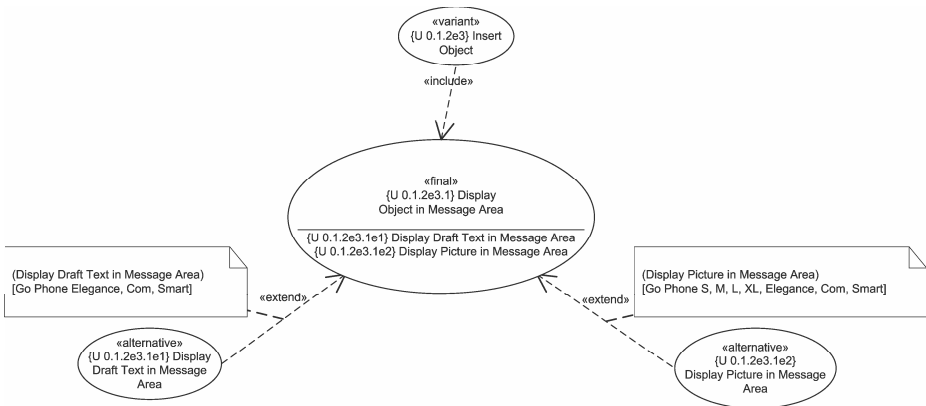


Fig. 9. *Insert Object* decomposition and included use case's extension points

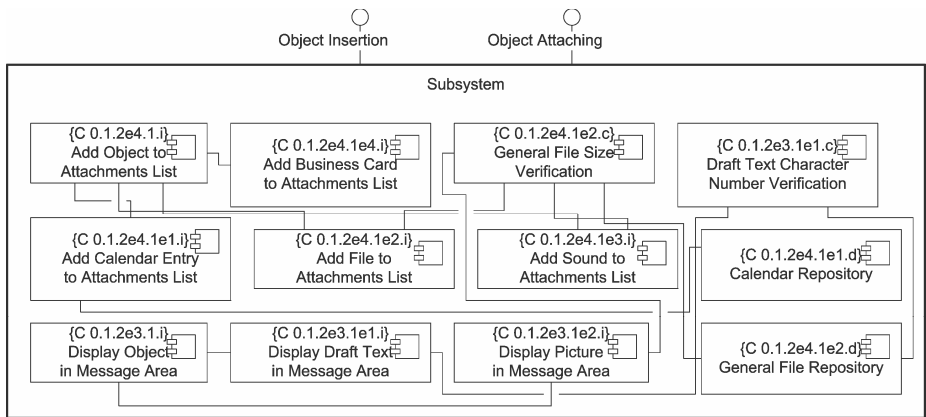


Fig. 10. Component diagram resulting from the recursive execution of the 4SRS

7 Conclusions

The 4SRS considered refinement in past works, yet, so far, there hasn't been any experimentation on the systematization of its recursive execution. The relevance of the exercise we presented in this paper resides on the demonstration that a modeling method is capable of dealing with the refinement of product line logical architectures systematically, which gains acute importance in the context of a high number of functionalities. With its extension, the 4SRS provides, now, for specific guidelines for modeling functional requirements when refining product line architectures. Besides the extension of a modeling method, we have worked on filtering and collapsing techniques in order to prepare the recursive execution of the modeling method over product line architectures. We were guided by the belief that not all components of product line architectures need refinement. We strongly believe that stepwise methods for the modeling of product line architectures would have a significant impact on the execution effort of some of their steps, if refinement was taken into account.

References

- [1] Kruchten, P.: Architectural Blueprints - The "4+1" View Model of Software Architecture. *IEEE Software* 12, 42–50 (1995)
- [2] Bragança, A., Machado, R.J.: Deriving Software Product Line's Architectural Requirements from Use Cases: An Experimental Approach. In: 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005), Rennes, France. TUCS General Publications (2005)
- [3] Bragança, A., Machado, R.J.: Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification. In: 10th International Software Product Line Conference (SPLC 2006), Baltimore, Maryland, USA. IEEE Computer Society, Los Alamitos (2006)
- [4] Machado, R.J., Fernandes, J.M., Monteiro, P., Rodrigues, H.: Transformation of UML Models for Service-Oriented Software Architectures. In: 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS 2005), Greenbelt, Maryland, USA. IEEE Computer Society, Los Alamitos (2005)
- [5] Muthig, D., John, I., Anastasopoulos, M., Forster, T., Dörr, J., Schmid, K.: GoPhone - A Software Product Line in the Mobile Phone Domain. Fraunhofer IESE, IESE-Report No. 025.04/E (March 5, 2004)
- [6] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, Carnegie Mellon University, Technical Report (1990)
- [7] Atkinson, C., Bayer, J., Muthig, D.: Component-Based Product Line Development: The Kobra Approach. In: 1st Software Product Line Conference (SPLC 2000), Denver, Colorado, USA. Kluwer Academic Publishers, Dordrecht (2000)
- [8] Weiss, D.M.: Commonality Analysis: A Systematic Process for Defining Families. In: van der Linden, F.J. (ed.) *Development and Evolution of Software Architectures for Product Families*. LNCS, vol. 1429, pp. 214–222. Springer, Heidelberg (1998)
- [9] Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology* 11, 215–255 (2002)
- [10] Greenfield, J., Short, K.: *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis (2004)
- [11] Egyed, A., Mehta, N., Medvidovic, N.: Software Connectors and Refinement in Family Architectures. In: 3rd International Workshop on Development and Evolution of Software Architectures for Product Families (IWSAPF-3), Las Palmas de Gran Canaria, Spain. Springer, Heidelberg (2000)
- [12] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 355–371 (2004)
- [13] Gomaa, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, Boston (2004)
- [14] Eriksson, M., Börstler, J., Borg, K.: Software Product Line Modeling Made Practical. *Communications of the ACM* 49, 49–53 (2006)
- [15] Bayer, J., Flege, O., Gacek, C.: Creating Product Line Architectures. In: 3rd International Workshop on Development and Evolution of Software Architectures for Product Families (IWSAPF-3), Las Palmas de Gran Canaria, Spain. Springer, Heidelberg (2000)