Adopting Computational Independent Models for Derivation of Architectural Requirements of Software Product Lines

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

The alignment of the software architecture and the functional requirements of a system is a demanding task because of the difficulty in tracing design elements to requirements. The 4SRS (<u>F</u>our <u>S</u>tep <u>R</u>ule <u>S</u>et*) is a UML based model driven method for single system development which provides support to the software architect in this task. This paper presents an evolution of the 4SRS method aimed at product lines. In particular, we describe how to address the transformation of functional requirements (use cases) into component based requirements for the product line architecture. In order to achieve this, we have extended the 4SRS method to explicitly handle variability. This evolution of 4SRS is based on an extension of the UML-F profile proposed by the authors. The UML-F profile provides UML notational extensions required to model variability. We present our approach in a practical way and illustrate it with an example.*

1. Introduction

The alignment of the software architecture and the functional requirements of a system is a demanding task. The conceptual gap that exists between the problem domain and the solution domain is very significant for the majority of the software projects. When this happens, it becomes difficult to co-relate requirements specifications and design decisions. The software architecture can rapidly become unsynchronized with the specified requirements of the system. To address this problem it is necessary to keep links between elements of the different levels of development. Achieving this without proper methodological and tool support is a daunting task. Model-driven development is a promising approach since models are treated as first class software artifacts, as traditional development does with code.

Model-driven methods are still a research topic. Much of the actual effort is on supporting techniques for transformation of models. One example is QVT[1], an OMG initiative to standardize model transformations in MDA. Also, reports of model-driven approaches tend to focus on transformations and are usually applied to design and implementation models, and usually do not include requirement and analysis models. Nonetheless, requirements specification and analysis are crucial activities of all software development processes. They drive the design of the system's architecture. As such, they should be integrated into model-driven methods.

Product lines are also another concept that is gaining popularity among the software industry. In this paper we will present a proposal to extend a model-driven method called 4SRS (*<u>F</u>our <u>S</u>tep <u>R</u>ule <u>S</u>et*) [2] so that it can be used to derive the architectural functional requirements of a product line from its requirements. The 4SRS method is based on UML [3]. The method applies transformational rules in order to derive a high-level architecture from use case requirements. For modeling variability within 4SRS we propose the adoption of the UML-F profile [4]. Since the UML-F original focus was on class diagrams this paper presents some extensions to address variability modeling in analysis diagrams. This paper also describes rules that can be used to transform requirements models into architectural models in a way that preserves variability and can be used without previous extensive knowledge of the domain.

The remainder of this paper is structured as follows. Section 2 briefly describes the 4SRS method and also presents our demonstration case: a small product line for libraries. Section 3 describes how requirements are modeled in 4SRS through use cases and activity diagrams. In Section 4, we describe how use case realizations are used to derive the architectural requirements of a product line. We address the global logical architecture of the system in Section 5. In Section 6 we discuss our approach and related work. Section 7 concludes the paper.

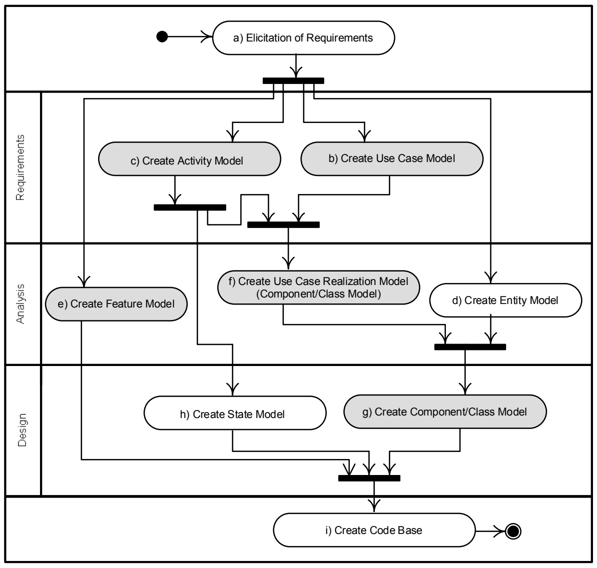

Fig. 1. Major activities of the 4SRS Method

2. The 4SRS Model-Driven Method

4SRS is a method aimed at supporting the transition from system requirements to software architectures and design elements. The method is essentially based on mapping UML use case diagrams into UML component diagrams. It uses an approximation by which a use case is realized by a collaboration of components of three kinds: interface, control and data; similarly as suggested in RSEB (Reuse-driven Software Engineering Business) [5]. After this initial transformation, a series of steps with rules are proposed to transform the initial component model into a coherent component model that is compliant with the requirements. Basically, in each step, a set of refactoring rules are applied that modify the initial component model by grouping, splitting or discarding components. Some of these rules can be automated, others depend on human intervention.

Method

Fig. 1 presents an overview of possible activities of model-driven methods aimed at the development of product lines. These are also the activities that compose the 4SRS method. In this paper we will only cover the activities that are marked in gray (*b*, *c*, *e*, *f* and *g*). The next section describes how the 4SRS method formalizes use case behaviors through activity diagrams. This enables 4SRS to capture functional requirements in a precise way. These activity diagrams also capture a very fundamental concept of product lines: variation points. The next sections also briefly discuss feature diagrams and how the concepts of these three kinds of diagrams relate to each other. For clarity reasons, some details are left out of the diagram of Fig. 1. For instance, activities *b*, *c*, *d* and *e* can be done in parallel but are not independent, they require coordination between them.

The 4SRS method is based on use cases and transformation rules that create other elements. As elements are created, their names are prefixed with a codification enclosed in brackets that is used to guarantee uniqueness and facilitate visual identification. The 4SRS transformation rules are applied through several of the activities presented in Fig. 1.

Briefly, the steps of the method are:

1. Component Creation
2. Component Elimination
    2.i Use Case Classification
    2.ii Local Elimination
    2.iii Component Naming
    2.iv Component Description
    2.v Component Representation
    2.vi Global Elimination
    2.vii Component Renaming
3. Component Packaging and Aggregation
4. Component Association

The method has been applied (and adapted) in several cases, from e-government [2] to protocol processing applications [6]. Experimental work on adapting the method to handle variability in the context of product lines has also been presented in the past [7]. From these previous works we have identified two fundamental issues regarding UML that require further clarification in order to fully integrate use case models into the 4SRS method. These issues are the semantics of the use case relationships (include, extend, and generalization) and the formalization of use case behaviors. These two issues are addressed by the extension to the UML metamodel proposed in previous work of the authors [8]. The major extension proposed was that Extend relationships may require rejoin points that are different from the original extension point. In the next section, we briefly present the 4SRS metamodel and discuss these issues.

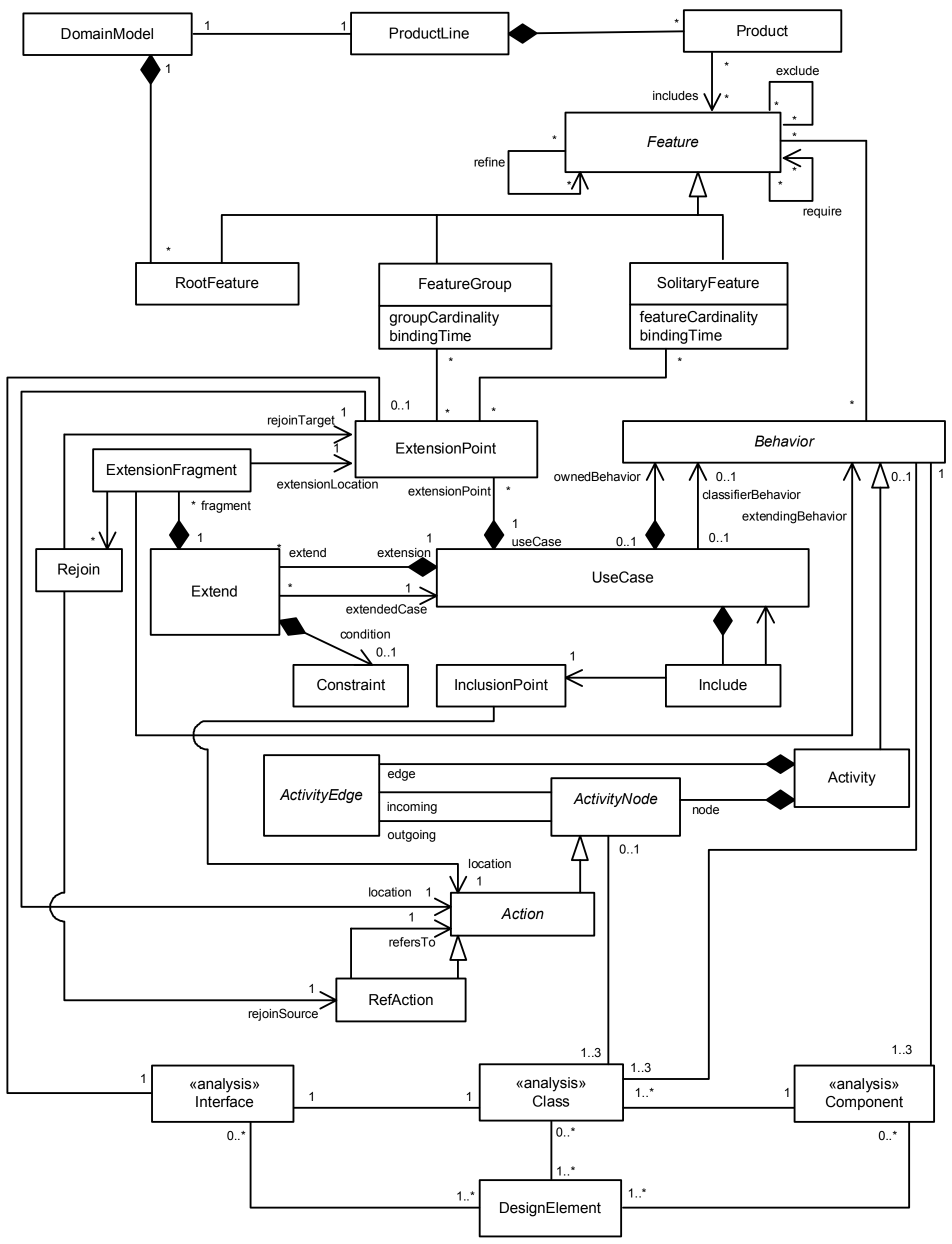

Fig. 2. Excerpt of the 4SRS metamodel

Metamodel

Fig. 2 presents an excerpt of the 4SRS metamodel. The figure contains only the necessary elements to support the discussion of the topics of this paper. From the figure we observe that this metamodel is an adaptation of the UML metamodel. This essentially relates to the fact that the UML *extend* relationship lacks the support for rejoin points, as indicated previously. As advocated in another paper of the authors [8], alternative behavior need the *rejoin* concept to be totally specified. So, in the 4SRS metamodel, the new element *ExtensionFragment* adds this support to the original *extend* relationship of UML.

The formalization of use case behavior by means of activities is also depicted in Fig. 2. For each use case behavior there is an activity. Extend and rejoin points of use cases trace directly to nodes of activities.

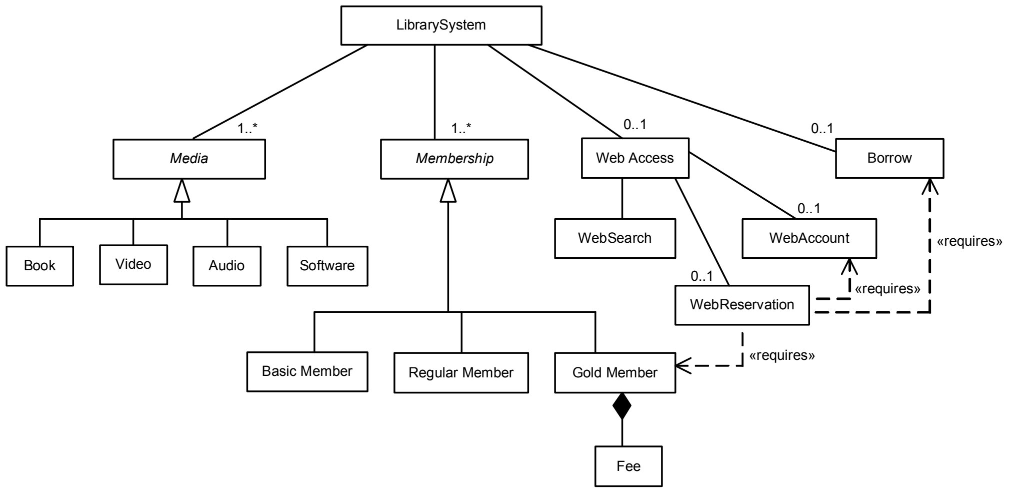

Fig. 3. Feature diagram for a library product line, following notation proposed in [9]

The advantage of adopting a model-driven approach, if it is supported by proper metamodeling tools, is that it is possible to add or adapt the metamodel. One example of such adaptation is the *ExtensionFragment* element. Another example of altering the UML metamodel is the added support for feature diagrams that is also depicted in Fig. 2. Although feature diagrams [10] are not part of the UML metamodel, they are a crucial artifact of product lines. They model the characteristics of a product line and how they relate to each other. A selection of the features represents a particular application of the product line. As such, features should relate to the requirements of the product line. Fig. 2 also presents these relationships.

UML-F

UML-F was proposed as a UML profile for frameworks [4] and later support was added to product lines [11]. With this profile it is possible to annotate UML elements with stereotypes that properly model variability. Unfortunately, the original UML-F profile only covers design elements. The profile lacks support for requirements and analysis models. As such, 4SRS had to extend the UML-F profile to include support for requirements and analysis models. Table 1 summarizes these stereotypes and informally defines their semantics.

Demonstration Case: Library Product Line

To demonstrate the 4SRS method, and particularly how it can be used to derive the architectural requirements of a product line, we will use a library product line. Suppose there is a software company that is planning to provide software management applications for libraries. Such company could adopt a product line approach. It could develop products with different features to address the specific needs of its customers. It could also plan on release versions of its products with different features at different moments. Fig. 3 presents a possible feature diagram for such a product line. We follow a notation for feature diagrams that is based on the one proposed in [9].

Since features usually represent capabilities or characteristics of systems they have a close relationship with use cases. For instance, Gomaa argues that a feature is realized by one or more use cases [12]. In the FeatuRSEB method [13], features and use cases are integrated. In this method, use cases help identify and construct the feature model. The 4SRS method adopts a similar approach. As Fig. 1 shows, the feature model is usually built after the use case model. In this paper we will not describe in detail this process, nor will we present the global use case model for the library product line.

3. Modeling Requirements with Use Cases and Activity Diagrams

UML use cases are very useful in capturing requirements because of their simplicity but, as discussed in the previous section, they also have some informal characteristics that difficult their adoption in model-driven methods. To address these issues, 4SRS extends the UML use case metamodel and adopts *Activities* to formally specify use case behavior. In 4SRS, for each use case behavior there is an activity diagram.

Table 1. Summary of UML-F stereotypes and their meanings

Stereotype	Applies to element	Description
variant	UseCase	Indicates that the behavior of the use case can vary.
mandatory optional alternative	UseCase	Classifies use cases according to their *inclusion* in the product line.
inclusion_point	Action	Indicates that the *Action* is an inclusion point for the classifierBehavior of the included use case.
extension_point	Action	Indicates that the behavior of the use case can be extended at the *Action*.
vp	ExtensionPoint	Indicates that the *ExtensionPoint* is a variation point of the product line.
template	«analysis»Component «analysis»Class DesignElement	Indicates an element which behavior is affected by variants that relate to a hook (based on [11]).
hook	«analysis»Interface DesignElement	An element that represents (or contains) a location where variations occur, i.e., a variation point (based on [11]).
rejoin_point	RefAction	Indicates that this *RefAction* rejoins the flow at the referenced *Action* of the base behavior. Attributes: *Moment* (*before* or *after*).
application	DesignElement	Indicates that the *DesignElement* relates to a specific application of the product line (based on [11]).
framework	DesignElement	Indicates that the *DesignElement* is global to all applications of the product line (based on [11]).
variable	Method	Indicates that the behavior of the method varies (based on [4]). Attributes: *Instantiation* (*dynamic* or *static*).
extensible	Class	Indicates that new methods can be added to the class (based on [4]). Attributes: *Instantiation* (*dynamic* or *static*).
incomplete	Generalization	Indicates that the generalization set can be incomplete, i.e., it is possible to add new classifiers to the set (based on [4]). Attributes: *Instantiation* (*dynamic* or *static*).

When use cases of the domain are identified, their behavior is modeled by activity diagrams. This is not so different than the traditional way of describing use case behavior by natural language, such as in [14]. Basically, each step in a text description of a use case is modeled as an *Action* node in the activity diagram. Sequence diagrams can be a helpful tool in modeling diverse use case scenarios that together describe the global behavior of a use case. They also can help when building the activity diagrams.

Use case relationships are discovered during use case modeling. The informal «include» and «extend» relationships become formal as they are modeled in 4SRS, since they relate *Action* nodes and use case elements in a precise way (see Fig. 2). Common use cases for diverse applications become mandatory use cases of the product line. Optional and alternative use cases will participate in «extend» relationships or, less commonly, in «include» relationships[1]. As such, during this process of modeling use cases, we are also identifying features of the product line. However, feature diagrams should not become direct mappings of use cases. For instance, all mandatory use cases could relate to a single root feature. In 4SRS, establishing relationships between features and use cases is a human task since it requires human decision. Nonetheless, it is possible for a tool to automate part of this process and to suggest operations on the models based, for instance, on the stereotypes presented on Table 1.

[1] Since an *including* use case is aware of the existence of the *included* use case some precautions are necessary when modeling variability with the «include» relationship. This is not the case for *extending* use cases, since the *extended* use case is not aware of being extended.

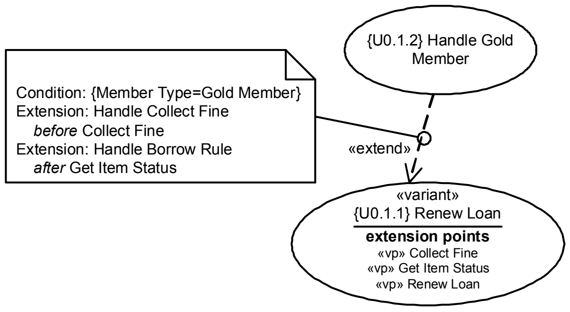
{U0.1.2} Handle Gold Member
Condition: {Member Type=Gold Member}
Extension: Handle Collect Fine before Collect Fine
Extension: Handle Borrow Rule after Get Item Status
«extend»
«variant»
{U0.1.1} Renew Loan
extension points
«vp» Collect Fine
«vp» Get Item Status
«vp» Renew Loan

Fig. 4. An «extend» relationship between use cases *{U0.1.2} Handle Gold Member* and *{U0.1.1} Renew Loan* (Following proposed extension to UML notation)

Fig. 4 presents the optional use case *{U0.1.1} Renew Loan* of the library product line (this use case is only present in an application of the product line if the feature *Borrow* is selected). This use case is extended by the use case *{U0.1.2} Handle Gold Member*. This extension was created since different kinds of membership impose different behavior in the system. Extension points usually give support to optional or alternative features. Since, in this case, an application of the library product line can support one or more member types, the feature *Membership* (see Fig. 3) is modeled as a *GroupFeature*. As such, the extending use case *{U0.1.2} Handle Gold Member* becomes a realization of the feature *Gold Member* (one of the composing features of *Membership*). The *Membership* feature (a *GroupFeature*) relates to the extension points *Collect Fine*, *Get Item Status* and *Renew Loan* of the use case *{U0.1.1} Renew Loan*.

Similarly to this simple case, it is possible to relate use cases and features as the requirements models are built. Other possible approach is to build the feature model after the use case model. The two approaches are possible in 4SRS and both compatible with the goal of keeping relationships (*links*) between use case model elements and feature model elements.

In fact, since use case behavior is modeled by activity diagrams, a trace can be made from elements of the activity diagrams to features, through use case elements. Fig. 5 presents the activity diagrams for the main behavior of use case *{U0.1.1} Renew Loan* and for two alternative behaviors of the extending use case *{U0.1.2} Handle Gold Member*. The activity nodes marked with the stereotype «extension_point» relate to the correspondent use case extension points with the same name. And, as we saw, these extension points relate to the *Membership* feature. Similarly, the alternative behaviors of use case *{U0.1.2} Handle Gold Member* that extend *{U0.1.1} Renew Loan* at the previous mentioned extension points should be related to one of the sub-features of *Membership*. In this case they relate to the sub-feature *Gold Member*.

Nodes of the 4SRS activity diagrams are marked with additional information that is used to support the 4SRS transformation rules. The stereotypes «interface», «control» and «data» are used to classify each node regarding its semantic role in the system. For instance, the node *Find Loan* represents a *data* operation in the system. We also use the concept of *partitions* to model 'who' has the responsibility for the operation of the node or is the major 'actor' of the node. For instance, the *System* is responsible for the node *Find Loan*.

When modeling activities, one very important aspect is the specification of object flows, i.e., input and output pins of the nodes. As they are specified, data types and associations are discovered. These are of great value since they provide input when creating the entity model of the domain (activity *d* of Fig. 1).

4. Capturing Functional Architectural Requirements with Use Case Realizations

The adoption of activity diagrams for modeling use case behavior results in a precise specification of the functional requirements of the product line. A use case realization is usually modeled by a group of analysis classes that *collaborate* to perform the use case behavior. They represent the first step in the transition from the problem space to the solution space. As such, they should be the primary (eventually the only) input for the software engineer as he/she designs the product line.

The first task of the design is the specification of the architecture of the product line, i.e., the collection of computational components of the product line and the interactions between these components. These elements can be essentially derived based on the input of use case realizations. Other requirements may also influence the architecture. For instance, the architectural *style* of a product line can be influenced by the specific run-time platform; the topology of the hardware; etc. In this paper we will only address the functional requirements for the architecture.

Traditionally, the specification of use case realizations is a very creative task. It requires a lot of experience from the software engineer, as he/she identifies the classes that realize the use case from the use case textual description. However, even a very experienced software engineer can misinterpret the requirements or forget some specification. What 4SRS proposes is the automation of this task.

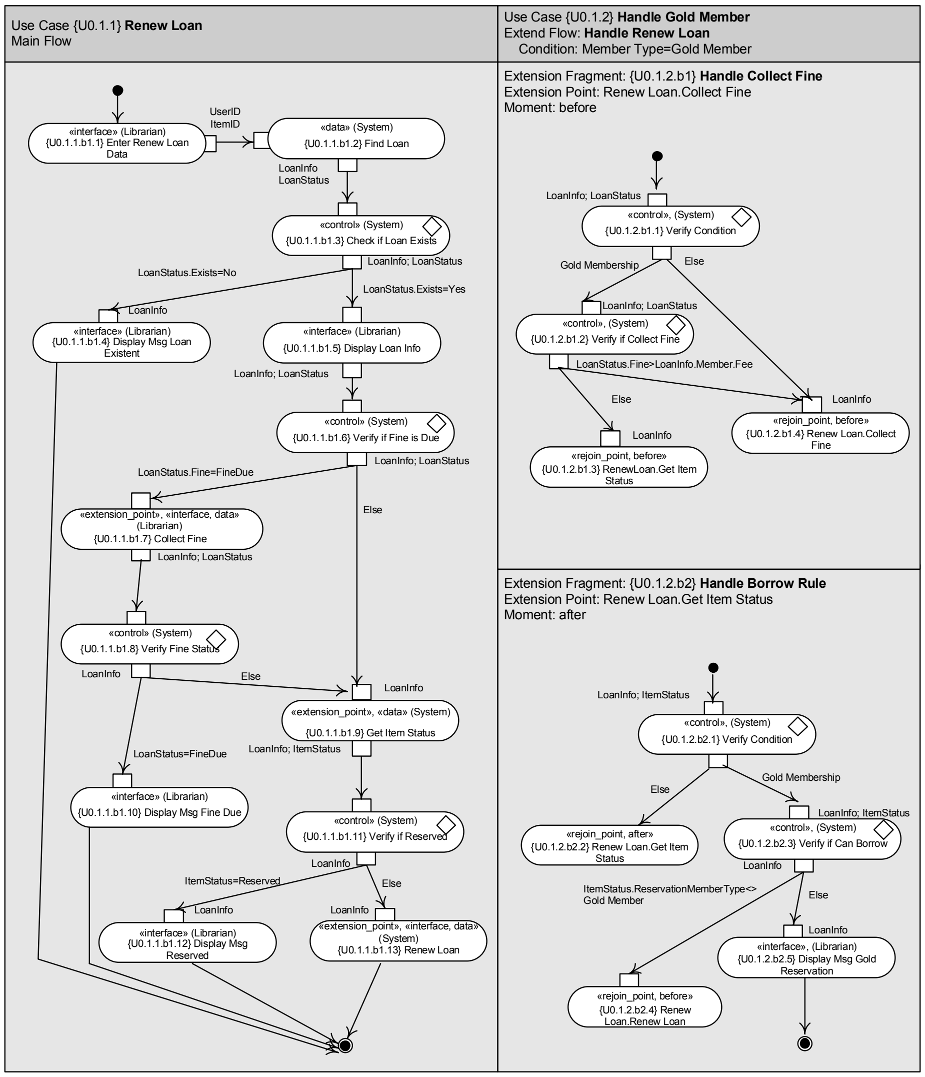
Use Case {U0.1.1} Renew Loan
Main Flow
Use Case {U0.1.2} Handle Gold Member
Extend Flow: Handle Renew Loan
Condition: Member Type=Gold Member
«interface» (Librarian) {U0.1.1.b1.1} Enter Renew Loan Data
UserID
ItemID
«data» (System) {U0.1.1.b1.2} Find Loan
LoanInfo
LoanStatus
«control» (System) {U0.1.1.b1.3} Check if Loan Exists
LoanInfo; LoanStatus
LoanStatus.Exists=No
LoanStatus.Exists=Yes
LoanInfo
«interface» (Librarian) {U0.1.1.b1.4} Display Msg Loan Existent
«interface» (Librarian) {U0.1.1.b1.5} Display Loan Info
LoanInfo; LoanStatus
«control» (System) {U0.1.1.b1.6} Verify if Fine is Due
LoanInfo; LoanStatus
LoanStatus.Fine=FineDue
Else
«extension_point», «interface, data» (Librarian) {U0.1.1.b1.7} Collect Fine
LoanInfo; LoanStatus
«control» (System) {U0.1.1.b1.8} Verify Fine Status
LoanInfo
Else
LoanInfo
«extension_point», «data» (System) {U0.1.1.b1.9} Get Item Status
LoanInfo; ItemStatus
LoanStatus=FineDue
«interface» (Librarian) {U0.1.1.b1.10} Display Msg Fine Due
«control» (System) {U0.1.1.b1.11} Verify if Reserved
LoanInfo
ItemStatus=Reserved
Else
LoanInfo
LoanInfo
«interface» (Librarian) {U0.1.1.b1.12} Display Msg Reserved
«extension_point», «interface, data» (System) {U0.1.1.b1.13} Renew Loan
Extension Fragment: {U0.1.2.b1} Handle Collect Fine
Extension Point: Renew Loan.Collect Fine
Moment: before
LoanInfo; LoanStatus
«control», (System) {U0.1.2.b1.1} Verify Condition
Gold Membership
Else
LoanInfo; LoanStatus
«control», (System) {U0.1.2.b1.2} Verify if Collect Fine
LoanStatus.Fine>LoanInfo.Member.Fee
Else
LoanInfo
LoanInfo
«rejoin_point, before» {U0.1.2.b1.4} Renew Loan.Collect Fine
«rejoin_point, before» {U0.1.2.b1.3} RenewLoan.Get Item Status
Extension Fragment: {U0.1.2.b2} Handle Borrow Rule
Extension Point: Renew Loan.Get Item Status
Moment: after
LoanInfo; ItemStatus
«control», (System) {U0.1.2.b2.1} Verify Condition
Else
Gold Membership
LoanInfo; ItemStatus
«rejoin_point, after» {U0.1.2.b2.2} Renew Loan.Get Item Status
«control», (System) {U0.1.2.b2.3} Verify if Can Borrow
LoanInfo
ItemStatus.ReservationMemberType<> Gold Member
Else
LoanInfo
«interface», (Librarian) {U0.1.2.b2.5} Display Msg Gold Reservation
«rejoin_point, before» {U0.1.2.b2.4} Renew Loan.Renew Loan

Fig. 5. Activity diagrams for *{U0.1.1} Renew Loan* and *{U0.1.2} Handle Gold Member*

The automatic creation of use case realizations in 4SRS is possible since use case behavior is totally specified by activity diagrams. The annotations made on the activity diagrams (as described in the previous section) also support this automatic transformation. The basic idea is that each node of the activity diagrams gives origin to up to three analysis interfaces and analysis classes that implement the interfaces. The interfaces that originate from nodes have one method that is based on the input and output pins of the node.

Fig. 6 and Fig. 7 presents the use case realization of, respectively, use case *{U0.1.1}Renew Loan* and use case *{U0.1.2} Handle Gold Member*. Since we are addressing design at an architectural level we find that component diagrams are more adequate than class diagrams to model use case realizations. As such, in 4SRS use case realizations are component diagrams. Each use case gives origin to up to three analysis components. Each one is composed by the classes and interfaces that resulted from the transformation of the activity nodes. For instance, for the use case realization of *{U0.1.1} Renew Loan*, the *Collect Fine* node gives origin to two analysis classes: *{c0.1.1b1.7.i}CollectFine* and *{c0.1.1b1.7.d}CollectFine* because the correspondent node was annotated with stereotypes *interface* and *data*. Based on the *extend* relationship that exists between the two use cases, it is also possible to automatically annotate the use case realization elements with the *hook* and *template* stereotypes of UML-F.

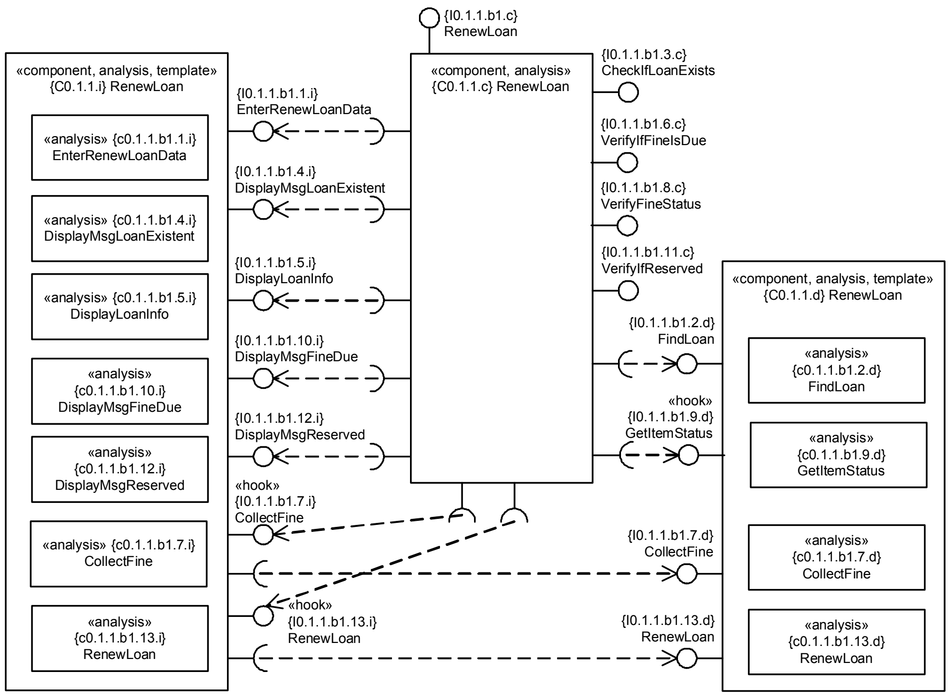

Fig. 6. Use case realization diagram for *{U0.1.1} Renew Loan* (filtered view)

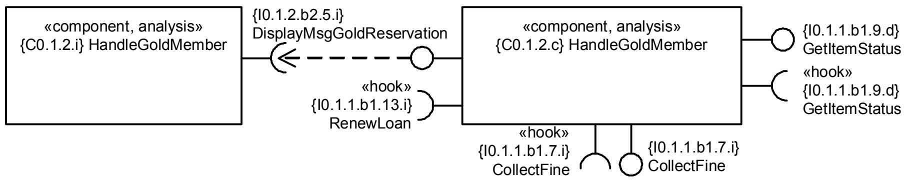

Fig. 7. Use case realization diagram for *{U0.1.2} Handle Gold Member* (filtered view)

The creation of use case realizations in 4SRS is a totally automated task. As such, all the generated elements are linked to their origins. This makes it possible to trace, for instance, a feature to its realization elements. Another advantage of this approach is that use case realizations can be easily transformed into executable code for a specific language or platform. Thus, use case realizations can also provide simple prototypes of the product line that can be of great help for the user validation of use cases.

As we saw, the 4SRS approach to use case realizations can be totally automated. One could argue that a complex use case could give origin to a complex use case realization. This is true, but eventually this would also probably happen if the creation of use case realizations were not automated.

Besides the previously discussed characteristics of 4SRS use case realizations, the fundamental value of the method is that as a result of a simple approach, the product line engineer is able to reason with precise functional architecture requirements.

5. Logical Architecture

Each use case realization provides a partial view of the product line. It is necessary to create a global model of the architecture of the product line. In 4SRS this global architecture is based on the use case realizations and represents an integration of them. In contrast with the creation of the use case realizations, the creation of the architecture is a human based task.

In this task, the product line engineer has to transform the analysis elements that resulted from the use case realizations into design elements. Each analysis element gives origin to a new design element or is incorporated into an existing one. For instance, all «interface» analysis components that are related to the librarian role can be incorporated into the *LibrarianUI* design component.

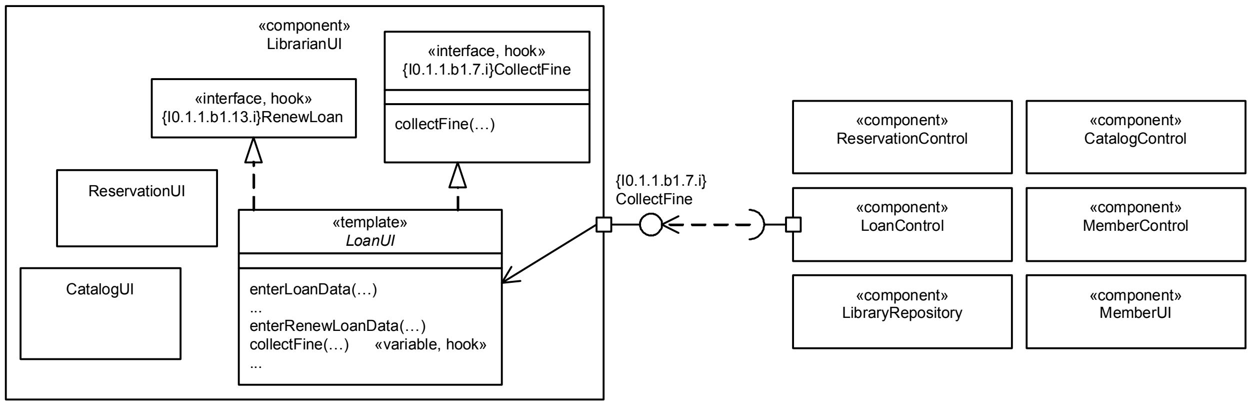

Fig. 8. Architectural logical view showing *{I0.1.1.b1.7.i}CollectFine* connecting the *LibrarianUI* and *LoanControl* components (filtered view)

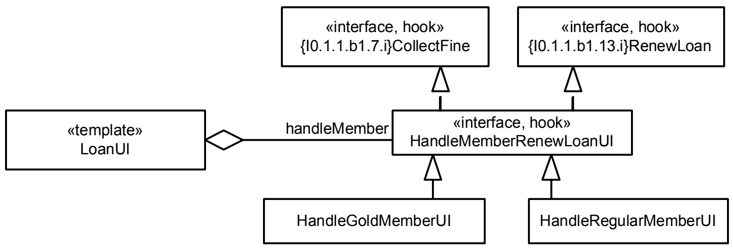

Fig. 9. Applying the abstract factory design pattern to realize the variability point of the *collectFine* method of the *LoanUI* class

Similarly, analysis classes give origin to new design classes or are incorporated into existing design classes. In Fig. 8, we can see that the analysis class *{c0.1.1.b1.7.i}CollectFine* was incorporated into the design class *LoanUI*. The only method of the class *{c0.1.1.b1.7.i}CollectFine* becomes a part of the *LoanUI* class. To facilitate the trace to the original elements of the use case realizations, analysis interfaces are not transformed, i.e., analysis interfaces also exist in design. As the described transformations are performed the global architecture of the product line takes form. Associations between components become visible as matching required and provided interfaces are transformed from analysis to design. Variability points identified in the requirements phase are traceable to variability points in the architecture of the product line. For instance, the hook method *collectFine* of the *LoanUI* class originates from the *Collect Fine* extension point of use case *{U0.1.1} Renew Loan*.

As Fig. 1 shows, the design model is not based only on use case realizations. The entity model is also an input for the design model. Its classes also populate the design model.

The tasks described in this paper enable the creation of an initial version of the architecture of a product line that is traceable to requirements and incorporates all functional requirements (as they were modeled). Obviously, the design of a product line does not end with its architecture. More detailed design tasks are required to achieve the goal of activity *i* (see Fig. 1): the creation of executable code. One example of such design tasks is the use of design patterns, as the ones proposed initially by Gamma *et al.* [15].

Fig. 9 is an example of the result of applying the *abstract factory* design pattern to realize the variability point of the *collectFine* method of the *LoanUI* class. Originally (Fig. 8), the *hook* (variability point) and the *template* were at the same class. The *abstract factory* design pattern separates the *template* from the *hook*. This realization of the variability point supports changing variants at runtime. Such a design decision should be made in accordance with the requirements. In this case, for instance, the *bindingTime* attribute of the feature *Membership* should have the value *runtime*. If only static binding time was required, the creation of subclasses of *LoanUI* would be sufficient to support the variability point.

This topic is out of the scope of the paper. Nonetheless, these are all examples of how the resulting artifacts of 4SRS can be used to support detailed design activities.

6. Discussion and Related Work

Use case realizations are a technique used to help the transition from the problem domain to the solution domain. The 4SRS method also adopts this technique. The particularity of 4SRS is that its use case realizations can be totally automated. This is possible because use case behaviors are formally modeled with activity diagrams and also because of the adaptations made to the UML metamodel. These adaptations support the proper modeling of variability in all the activities of the method.

As a result of the 4SRS approach, it is possible to maintain traces between elements at different conceptual levels. In the case of product lines, use case requirements are traced to architectural requirements: use case variability is traceable into architectural elements; features are related to architectural elements.

The original 4SRS method has been applied to develop pervasive and embedded systems [2]. In the past, we have also presented an example of an experimental 4SRS extension to support embedded product lines [7]. Our actual work aims at broadening the scope of 4SRS so that it can be used generically.

Transforming requirements to analysis models is a very difficult task since the semantic gap between the problem and solution domain is usually very significant. Usually, methods take for granted that the analyst is a domain expert and will discover the architectural elements without difficulty or that the architectural elements can be easily discovered by simply transforming nouns into objects and verbs into operations. We can find this kind of approach in methods like RSEB [5, 13], PLUS [12] and Catalysis [16]. In real projects the transformation process is much more difficult. Our approach (as well as the one of the original 4SRS) has more similarities with methods which make the initial transformation based on objects and not classes, such as Fujaba [17]. With this kind of approach, use cases are first realized by object collaboration diagrams. Only after realizing all use cases with object collaborations we proceed to create class models. Our approach goes even further since the use case realizations are based on performing the actions of the activity diagrams essentially by *fragments* of analysis classes. These *fragments* must exist to realize the system but are only transformed into design elements after the initial architecture of the system is completed. With this approach we aim at facilitating the work of the analyst since it is not required prior domain knowledge or architectural experience to build the first complete architecture of the system. One relative drawback of our approach is that it requires metamodeling tools for the adaptation of the UML metamodel. Tool support for 4SRS is being developed experimentally with EMF [18] and GMF [19]. We plan on making this tool support publicly available in the near future.

7. Conclusions

In this paper we have presented an approach (based on the 4SRS method) to derive product line's architectural requirements from use cases. To achieve this goal, use case behavior must be formally specified using activity diagrams. We have discussed how this specification can be made. The UML metamodel does not support required functionalities of 4SRS. We have also presented how the UML metamodel can be extended to support the method. The paper also briefly presented how the UML-F profile is used to support the modeling of variability.

For the transition between requirements and analysis we demonstrated how the use cases and activities can be transformed into an initial global architecture of the system. Our approach innovates in the sense that this transformation is totally automated. We argue that this approach can defend analysts from taking poor initial architectural decisions, particularly when the problem domain is new and there is no prior experience.

The 4SRS model-driven method can not be totally supported by common UML tools, since it requires adaptations to the UML metamodel. Tool support requires the use of metamodeling tools. This may have impact in the cost of applying 4SRS or a similar approach. The possible increase in the cost of a project can be balanced by the increase in flexibility that such approach also brings. To be notice that product line approaches already require important setup effort. This initial cost is amortized in the future, as new applications of the product line are created. As such, we believe that 4SRS, and similar approaches, are well suited for product lines.

In the near future we will apply the method to two real cases of product lines. These cases will provide precise data on the percentage of automated activities and effort savings.

8. References

[1] "MOF QVT Final Adopted Specification," OMG, 2005, Available at http://www.omg.org/.
[2] R. J. Machado, J. M. Fernandes, P. Monteiro, and H. Rodrigues, "On the Transformation of UML Models for Service-Oriented Software," presented at ECBS International Conference and Workshop on the Engineering of Computer Based Systems, Greenbelt, Maryland, 2005.
[3] "formal/05-07-04 Unified Modeling Language Version 2.0: Superstructure," OMG, 2006, Available at http://www.omg.org/.
[4] M. Fontoura, W. Pree, and B. Rumpe, "UML-F: A Modeling Language for Object-Oriented Frameworks," presented at ECOOP 2000-Object-Oriented Programming Conference, 2000.
[5] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*: Addison Wesley Longman, 1997.
[6] Marcus Alanen, J. Lilius, I. Porres, and D. Truscan, "On Modeling Techniques for Supporting Model Driven Development of Protocol Processing Applications," in *Model Driven Software Development - Volume II of Research and Practice in Software Engineering*, vol. 2, S. Beydeda, M. Book, and V. Gruhn, Eds.: Springer-Verlag, 2005, pp. 305-328.
[7] A. Braganca and R. J. Machado, "Deriving Software Product Line's Architectural Requirements from Use Cases: an Experimental Approach," presented at 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software, Rennes, France, 2005.
[8] A. Braganca and R. J. Machado, "Extending UML 2.0 Metamodel for Complementary Usages of the «extend» Relationship Within Use Case Variability Specification," presented at SPLC 2006, Baltimore, Maryland, 2006.
[9] A. v. Deursen and P. Klint, "Domain-Specific Language Design Requires Features Descriptions," *Journal of Computing and Information Technology*, vol. 10, pp. 1-17, 2002.
[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study Technical Report," Software Engineering Institute, Carnegie Mellon University CMU/SEI-90-TR-21, 1990.
[11] W. Pree, M. Fontoura, and B. Rumpe, "Product Line Annotations with UML-F," presented at Software Product Lines - Second International Conference, SPLC 2, San Diego, 2002.
[12] H. Gomaa, *Designing Software Product Lines with UML*: Addison Wesley, 2005.
[13] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," presented at Fifth International Conference on Software Reuse, Victoria, Canada, 1998.
[14] A. Cockburn, *Writing Effective Use Cases*: Addison-Wesley, 2001.
[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
[16] D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach*: Addison-Wesley Professional, 1998.
[17] "Fujaba," University of Paderborn Software Engineering Group, 2006, available at http://wwwcs.uni-paderborn.de/cs/fujaba/index.html.
[18] "Eclipse Modeling Framework," Eclipse Foundation, 2006, available at http://www.eclipse.org/emf/.
[19] "Graphical Modeling Framework," Eclipse Foundation, 2006, available at http://www.eclipse.org/gmf/.