Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Hovsepyan *Stefan Van Baelen*
Yolande Berbers *Wouter Joosen*

Report CW 508, January 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Hovsepyan *Stefan Van Baelen*
Yolande Berbers *Wouter Joosen*

Report CW 508, January 2008

Department of Computer Science, K.U.Leuven

Abstract

This report presents the GReCCo approach to Aspect Oriented Modeling (AOM) using Generic Reusable Concern Compositions. GReCCo offers an AOM-based framework to promote and enhance the reuse of oblivious concern models. We focus on software design patterns, which represent complete solutions to recurring concern-specific problems. We have developed a prototype generic transformation engine written in ATL that can be used to compose two concern models specified in UML.

We first describe the GReCCo approach and the offered composition types. In the second part, we illustrate the GReCCo approach on a case study in the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology.

Keywords : AOM, obliviousness, model compositions, reusable concerns.
CR Subject Classification : I.2.11, I.2.8, I.2.1.

Generic Reusable Concern Compositions (GReCCo): Description and Case Study

Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers, Wouter Joosen

Katholieke Universiteit Leuven, Departement Computerwetenschappen, Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Aram.Hovsepyan, Stefan.VanBaelen, Yolande.Berbers, Wouter.Joosen}@cs.kuleuven.be

Abstract. This report presents the GReCCo approach to Aspect Oriented Modeling (AOM) using Generic Reusable Concern Compositions. GReCCo offers an AOM-based framework to promote and enhance the reuse of oblivious concern models. We focus on software design patterns, which represent complete solutions to recurring concern-specific problems. We have developed a prototype generic transformation engine written in ATL that can be used to compose two concern models specified in UML. We first describe the GReCCo approach and the offered composition types. In the second part, we illustrate the GReCCo approach on a case study in the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology.

1 Introduction

Aspect-Oriented Modeling (AOM) is a recent development paradigm that brings Aspect-Oriented Programming (AOP) to a higher abstraction level by using Model-Driven Development (MDD) techniques. Following an AOM approach, we can model certain concerns of a complete solution separately, and compose them using model weaving techniques. AOM is a particular approach aiming at modularizing and composing concerns within software models, and can be considered as a special case of an MDD approach.

If AOM is going to bring an increase of efficiency in software development, we need to make sure that it is possible to write concern models once and reuse them in different contexts. If one has to model a concern and write a composition transformation every time from scratch, the gains of AOM will be diminished substantially. However, there is a need to define criteria that make a given concern reusable.

In this paper we define and characterize reusable concern models. We provide several key criteria that we believe are crucial in enhancing concern model reusability. We have developed a graphical framework that can model a certain type of concerns and specify their composition using a composition model. We have a prototype generic composition engine implemented using model transformations in ATL [1], which can compose UML concern models.

In section 2, we define the key requirements that are necessary to improve concern reusability. We also comment why current AOM approaches come short in achieving some of these requirements. In section 3.2, we present our approach in details. We then describe the different composition mechanisms of the Generic Reusable Concern Compositions (GReCCo) approach. In section 4, we present a case study and apply our approach to it. In section 5, we evaluate how GReCCo improves reusability by tackling each requirement presented earlier. In section 6, we present some related work. In the last section, we conclude and outline future work.

2 Concern Reuse

There is no clear notion of reuse for concern models in MDD as opposed to other software engineering approaches. Hence, we need to define and characterize what makes a given concern model more or less reusable.

2.1 Requirements for reuse

We define a reusable concern model as an independent solution for a known concern-specific problem, which can be used in several contexts to produce required assets. Different contexts mean for instance different applications, projects, companies, domains, etc. However, using this definition we cannot measure how reusable a concern model is. This is why we introduce several more measurable qualities of a concern model, which we believe are most important in increasing reuse of concerns.

Obliviousness: The concerns, represented by their models, should be fully independently defined from other concerns. The concern composition - all the relevant parameters, mappings, and any extra weaving directives - should be specified in a separate composition model. This is also true for the primary model - a special concern that models the functional core of a given application. It should not be dependent on any other concerns, which might eventually be composed with it.

Composition symmetry: We should be able to compose concerns not only with the main application (primary model), but also with each other, independently from the primary model. This will allow the reuse of concerns within other concerns.

Interdependency management: There are many (often hidden) interactions between the different concerns, since they are not always completely orthogonal to each other. Such concern interactions can be classified to one of the following five categories: dependency, mutual exclusiveness, alternatives, conflict and mutual influence [2]. Hence, in order to be able to achieve safe reuse, it should be possible to declare a potential interdependency explicitely, and detect it when composing different concerns.

2.2 Current approaches

There are many AOM approaches, each pursuing a distinguished goal and providing different concepts as well as notations. We consider them categorized by the alignment to phases criteria introduced by Op de beeck et al. [3].

AOM approaches aligned to implementation Some AOM approaches have no high level modeling concept that maps the design to the concerns identified during requirements phase. Typically they provide only notation mechanisms for AOP-level concepts such as join points, pointcuts, advices, etc [4,5,6,7]. Even though this allows the modeling of a given concern in a certain manner, these approaches remain too close to the implementation level. Such approaches typically use UML to specify what we call the primary model. For aspect-specific concepts, a light-weight UML extension is usually used. This is advantageous in terms of tools, since UML profiles are supported by almost any UML tool. However, these approaches do not score high given our reusability criteria. Concerns are often implicitly dependent on the structure of the primary model, which means that obliviousness is only unidirectional. Almost all implementation aligned approaches support composition asymmetry. An asymmetrical composition typically models the primary model and the concerns which cut across it as separate entities. It is said that crosscutting concerns are composed with the primary concern. It is impossible to compose crosscutting concerns with each other. In addition, these approaches provide no means to declare and detect concern interdependencies.

AOM approaches independent from implementation Several other approaches [8,9,10] are implementation independent and provide higher-level mechanisms for concern modeling and composition. These approaches are by default more reusable, as they are more abstract in nature. They are based on UML metamodel extensions, however, none offers appropriate tool support yet. Even though most of them implicitly support concern reuse to some extent, they do not explicitely focus on reuse. None of these approaches provides means to declare and detect concern interdependencies. Moreover, most of these AOM approaches use template parameters that constrain the concern re(use), as they predefine the context in which the aspect can be used. This makes concerns implicitly dependent on a primary model, which must bind concrete elements to the template parameters.

In this paper we introduce a conceptual framework for representing concerns and specifying their compositions with other concerns, which improves the support for reuse by tackling each of the requirements listed above. In addition, we reuse existing work [11] that solves the problem of declaring and detecting concern dependencies on other concerns.

3 Generic Reusable Concern Compositions (GReCCo)

In this section, we first describe the general principle behind the GReCCo approach. Then we present the specifics of the composition of concern models. In addition, we discuss the problem of concern interactions and show how an existing solution can be plugged into our approach.

3.1 General Description

The GReCCo approach is used to compose concerns represented by their models. We decided to constrain the number of models to two in each composition step. We rep-

resent each composition step as the Greek letter upsilon (ϒ). The left and the right branches of the upsilon contain two concern models. In theory, we make a separation between a primary (or base) concern and other concerns. The primary concern represents the (functional) core of the application. The rest of the concerns represent a whether or not crosscutting concern that is modularized and at a latter stage composed with the primary concern. However, our approach is **symmetric** and in practice it is unimportant what kind of concern models we are dealing with.

Along with the two concern models we also provide a composition model that instructs the model transformator how the two models should be composed (fig. 1).

Concern Models 1 and *2* describe the structure of the concerns using UML class diagrams and may contain behavior specification using UML sequence diagrams. The *Composition Model*, which consists of UML class and sequence diagrams, specifies how the concern models are composed by defining all composition-specific parameters and their bindings. This assures the **obliviousness** of the two concerns. Since the *Composition Model* defines the model composition, it is not independent as opposed to the source models, i.e., it is useful only if the related models exist as well.

Elements that are not referenced in the *Composition Model* are copied to the *Composed Model.* The rest of the elements are modified by the composition engine according to the specification, which will be described in detail in the following section. From these three input models, using a generic composition engine, we can obtain a generated output *Composed Model* (fig. 1).

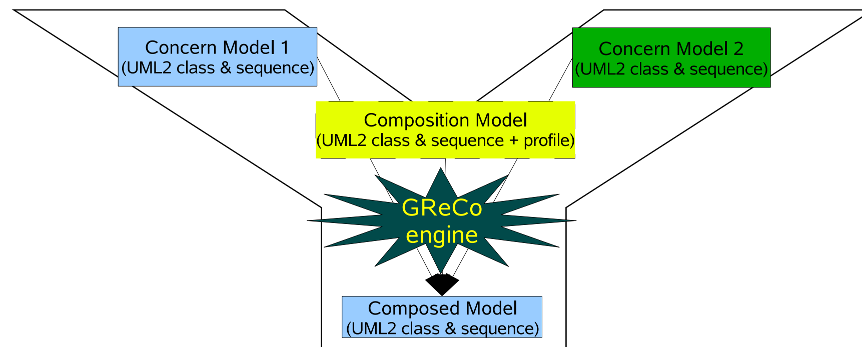

Fig. 1. General Approach

As mentioned previously, our approach is **symmetric** as it does not require one of the input concerns to be a primary model. It can be used in two scenarios, even though it does not make any difference for usage of GReCCo in practice.

- the first scenario realizes a symmetric approach where any two concern models can be composed (fig. 2),
- the second case is used to realize an asymmetric composition where a concern, represented by *Model 2*, is composed with a given primary model, represented by *Model 1* (fig. 3).

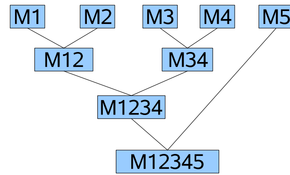

Fig. 2. Symmetric Composition

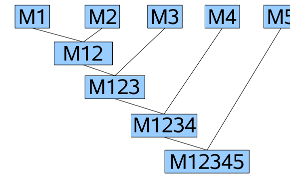

Fig. 3. Asymmetric Composition

Currently we have successfully applied our approach on design patterns by the Gang of Four (GoF) [12] and on security patterns [13].

3.2 Composition Specification

In order to compose two concern models, we need to specify the composition in the *Composition Model*. We consider the structural and behavioral concerns of the composition separately. Obviously, entities which are not involved in the composition specifications are copied to the *Composed Model* as they are, without any modifications.

Structure From the point of view of one concern model, we differentiate between three types of basic compositions that can happen to a single entity: (1) we can add a new entity, (2) we can modify the properties of an existing entity, and (3) we can remove an existing entity. As we are dealing with structural models, we discern between four types of UML entities: class, property, operation and association. When two concern models are composed, there are some additional compositions that can occur to the input element(s). Two elements from different models can be merged to obtain a single entity with the combined properties. This merging makes sense only for class entities. Finally, some concerns introduce roles and/or template parameters as semantic variation points, which should be instantiated by using concrete UML entities. Hence, it should be possible to indicate that a given model element is simply a template parameter, and must be instantiated by a concrete entity.

Add The composition of two concerns may need to add new entities. For instance, we might need to add an association between a class from each of the concern models. In order to do so, we simply add the entity to the mapping model and tag it with the UML stereotype $\ll add \gg$, which will indicate that this is a new entity that must be added to the composed model.

Modify During composition, we may need to modify some existing entities. For instance, we might need to rename an element because another one already exists with the same name. We specify this by marking the element to be modified with the $\ll modify \gg$ stereotype. We indicate the UML property, which needs to be modified, and

the modified value using a stereotype attribute (called a tag in UML 1.x). Each stereotype attribute will have a name, indicating the UML property that should be modified, a type, which is the same as the UML type of the property to be modified, and a value indicating the new value. If several UML properties of a given UML element need to be modified we will use several stereotype attributes and only a single $\ll modify \gg$ stereotype. For instance, if we wish to modify the name of a given class, we need to place a $\ll modify \gg$ stereotype on the class, and fill out the *name* stereotype attribute tag with the new name of the class.

Remove Due to the composition of concerns, certain entities may become unnecessary in the composed model. For instance, a concern may introduce an indirect association between certain entities, which are already connected directly in another concern. We realize the removal of elements by putting the ≪ *remove* ≫ stereotype on the element that needs to be removed.

Merge When two concern models are involved, we sometimes have to deal with elements that represent a different view on the same entity. During the concern composition, we need to merge these elements to obtain a single entity with combined properties. In order to merge two or more elements, we need to place a UML association between the elements and mark it with the ≪ *merge* ≫ stereotype. Even though different semantics can be associated with the merging, we have chosen to create a new element which is composed from the properties of the original two elements. By default, the name of the composed element is set to the concatenation of the names of the input elements. Conflicts such as name clashes, mutually exclusive properties, etc., should be resolved explicitly by using the modification strategy. Note that it is possible to merge more than two elements. In order to do this, we place binary ≪ *merge* ≫ associations between the classes.

Instantiate Often, a concern model represents a pattern that should be instantiated using elements from the other concern model. A pattern model contains a number of template elements (e.g., roles) that must be bound to concrete elements when the pattern is applied. In addition, a pattern model may contain a number of concrete elements that need to be introduced as new elements in the resulting model. As we aim for oblivious concerns, we model all pattern elements as a concrete ones. On the composition model, however, we will use the instantiation strategy, which tells the composition engine that a given concern element is a template that should be bound to a concrete one. In practice, instantiation is similar to merge with the only difference that all conflicts are resolved by taking the properties of the concrete element. In addition, the name of the composed element is kept the same as the name of the concrete element. To specify an instantiation, we need to place an $\ll instantiate \gg$ dependency link from the entity that we consider as a template to the concrete one.

Behavior We believe that UML sequence diagrams are a very suitable representation for describing the behavior of a given concern, which has its structure specified by a UML class diagram. Each sequence diagram represents a certain scenario. Scenarios,

which are completely independent of each other, are simply copied to the composed view. Given the structural mapping information, it is still impossible to devise an algorithm that automatically detects the independent scenarios. Hence, this choice is left to the person that performs the composition of the concerns.

We consider three possible mechanisms whenever two behavior scenarios are composed:

- the sequence of messages between the input behavior scenarios needs to be determined,
- a call from one input scenario is the same as a call from the other input scenario,
- a call or a set of calls from one of the input scenarios need to be replaced by a call or set of calls from the other input scenario.

In order to realize the first use-case, we introduce a notion similar to that of general ordering from the UML 2.0 specification [14] (p. 466). We introduce the general ordering as a directed binary relation between an event or a set of events enclosed in a fragment and another event or set of events. The resulting scenario defines a partial ordering of the input events where the direction of the relation indicates which events should come first. We use a dependency between the event(s) that should preceede another event and mark it with the ≪ *genordering* ≫ stereotype. The interpretation by the composition engine is that the dependency client should immediately preceed the dependency supplier. Events that are not involved in any general ordering relation are put in parallel combined fragments blocks.

For the second use-case we use a dependency between the two calls marked with the ≪ *merge* ≫ stereotype. If we do not use this dependency the call will be duplicated.

In the third scenario we use the notion of interaction fragments from the UML2 specification. The event(s) that are to be replaced as well as the replacing event(s) are put in combined fragments. We use the *loop* interaction operator for the fragments with a single iteration. Finally, we place a dependency marked by ≪ *replace* ≫ stereotype between the replacing and the to-be-replaced fragments.

We introduce a simple authentication concern, which consists of a *Client* and *Server* entities (fig. 4). The *Client* tries to login on the *Server*. In case of a successfull login the *Server* replies with an *OK* message. If the authentication failes, the *Server* logs the failed attempt and replies with a *try again* message (fig. 5).

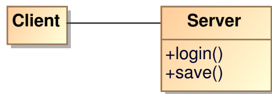

Fig. 4. Authentication Structure

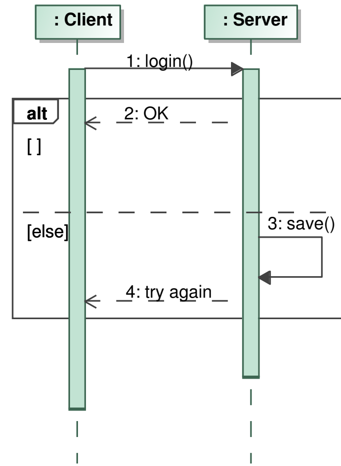

Fig. 5. Authentication Behavior

In addition, we introduce a simple logging concern, which consists of a *Client*, *Server* and *DBExchanger* entities (fig. 6). Every time the *Client* tries to login the *Server* asks the *DBExchanger* to log the event (fig. 7).

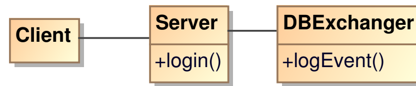

Fig. 6. Logging Structure

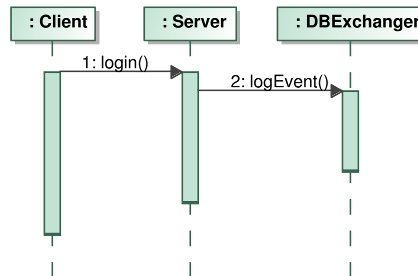

Fig. 7. Logging Behavior

Firstly, we specify the structural composition. We instantiate the *Client* as well as the Server entities from the two concerns. In addition we specify that the *login()* event from the authentication should be instantiated as a *login()* even from the logging concern (fig. 8).

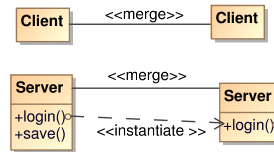

Fig. 8. Structural Composition

Then we specify the behavioral composition. We would like to obtain a composition that performs authorization and in case of a successfull authentication writes it to a log. If the authorization fails, the authorization concern itself is in charge of saving the failed attempt. The *login()* events from the two concerns are the same and we place a $\ll$ *merge* $\gg$ dependency link between them. In addition, we need to put the *logEvent()* event after the *OK* message. We specify this by placing a $\ll$ *genordering* $\gg$ dependency from *OK* to *logEvent()*. As a result we obtain a combined sequence scenario that performs both authentication and logging in case of a successfull authentication. Note that in case we need to use the logging scenario after the authentication independent from the outcome we need to place the $\ll$ *genordering* $\gg$ dependency from the *alt* fragment to the *logEvent()* (fig. 9).

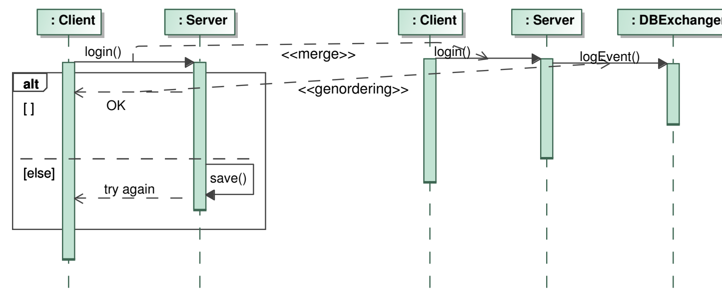

Fig. 9. Behavior Composition

3.3 Concern Interactions

Our approach can be used to obtain a system that is built by many concern compositions. Fig. 2 and 3 are examples of sample trees that can be obtained if we combine more than one concern model.

However, concern models are rarely completely orthogonal to each other, but can relate to each other in a variety of different ways. A concern can be involved in an arbitrary number of interactions with one or more other concerns. Sanen et al. [11] distinguish between five different classes of concern interactions: dependency, conflict, choice, mutex and assistance. They also provide a conceptual framework for describing

the relevant information about interactions between concerns that need to be captured. The approach comes with a Concern Interaction Acquisition (CIA) prototype system that can be used to describe and query particular concern interactions. Fig. 10 shows an overview of the expert system. Domain experts add expertise about interactions between concerns to an OWL ontology using the Protégé [15] environment. The acquired knowledge is automatically transformed into a set of Prolog rules. The GReCCo tool has to provide a specification of a certain concern composition to be investigated for potential concern interactions. Based on this list, a set of Prolog facts is generated that contains all the predicate definitions that describe the listed concerns. Both the Prolog rules and Prolog facts are fed into a Prolog engine that through reasoning can detect all the interactions that occur in the given concern composition. This list of interactions is presented back to the GReCCo tool.

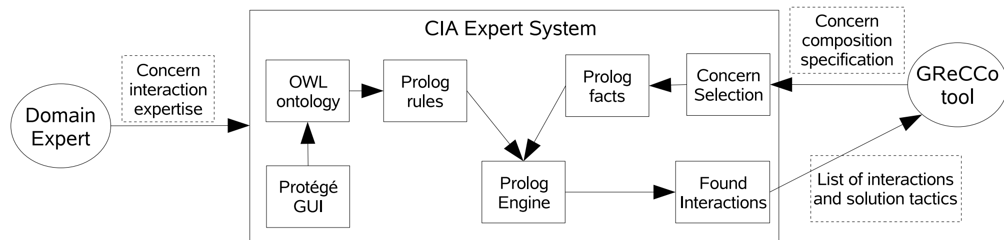

Fig. 10. Architecture of the CIA expert system

Our approach is currently constrained to the composition of only two concerns at a time. However, if we keep a composition history we could query about interactions with concerns that are already composed during the previous steps. This system is currently not incorporated in the GReCCo engine, however we believe that it is very much straightforward to implement it.

4 Case-Study

In this section, we present an application from the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the base part of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology described in the previous section.

4.1 Screening Application

Screening application represents an information system of a screening lab. Patients (*ScreeningSubject*) make an appointment to their radiographic pictures (*Screening*) taken by a *Radiographer*. Two different *Radiologists* perform a *Reading* of the radiographic screening. In case the reading results are the, same an automatic *Conclusion* is generated. Otherwise, a third reading takes place, whereafter the third radiologist creates a

final conclusion. In addition to the system itself, we have realized an additional client-server mechanism so that patients can consult their own data at home from, e.g., a web browser (*Client*). *RegistryService* offers a set of *ScreeningServices,* each having its own *id*. Fig. 11 presents a UML class diagram for the screening lab application. Fig. 12 represents the behavior of a service execution.

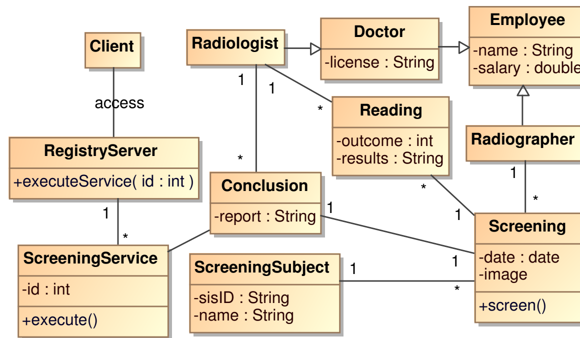

Fig. 11. Screening Lab Application Model

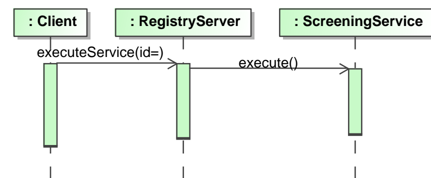

Fig. 12. Execute service

4.2 Application Firewall Pattern

The registry server, introducted in the previous section, must ensure that only authenticated and authorized clients may use a given service. A sound solution in this case would be to interpose an application-level firewall that can analyze incoming requests for the registry services and check them for authorization. A client can access a service of the registry server only if a specific policy authorizes it. Policies for each application are centralized within the *ApplicationFirewall* and they are accessed through a policy authorization point (*PAP*) (fig. 13 and 14).

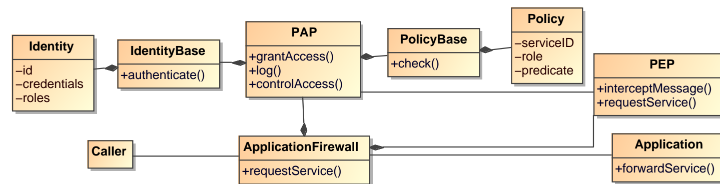

Fig. 13. Application Firewall Pattern

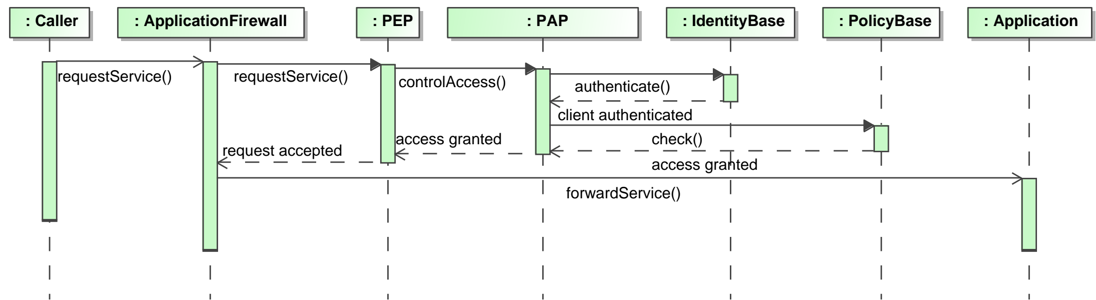

Fig. 14. Request Service

We want to compose the application firewall concern with the base. We follow the framework described in the previous section and define the composition model for the structure (fig. 15). We specify that *Application*, *Caller* and *forwardService()* are template elements that should be instantiated by *RegistryServer*, *Client* and *executeService()* elements respectively. In addition, we need to remove the direct association between *RegistryServer* and *Client* as the application firewall concern will introduce an indirect link between the two. For illustration purposes, we rename the *RegistryServer* to *Server* by placing a $\ll modify \gg$ stereotype on the class, using a tag *name* to indicate the new name.

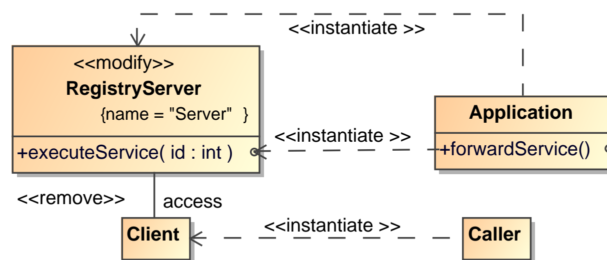

Fig. 15. Structural Composition Model

We want to obtain a composed behavior that actually changes the direct *executeService* call from *Client* to *RegistryServer* by an indirect *requestService* event. To specify this we place a $\ll replace \gg$ dependency link from *requestService* to the *executeService* event (fig. 16).

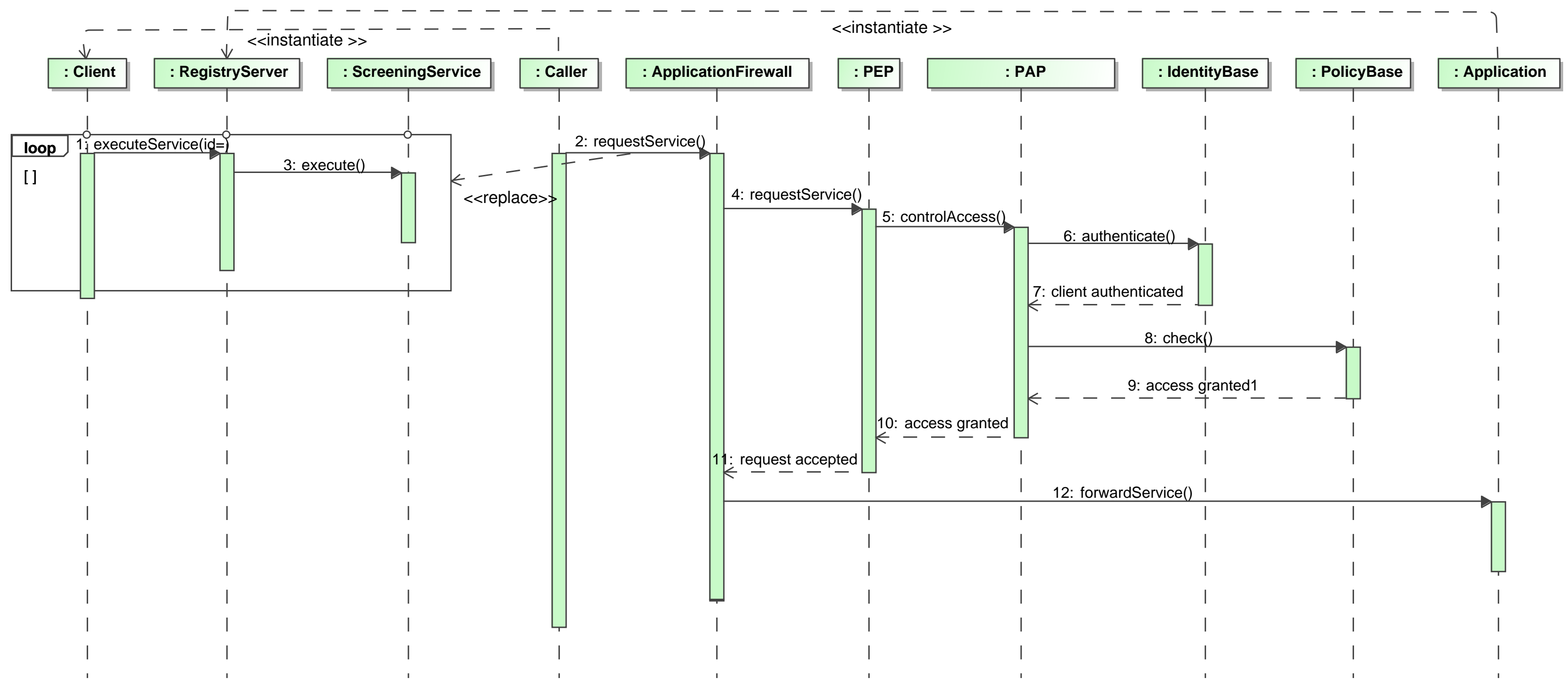

Fig. 16. Behavior Composition Model

The structure and behavior of the resulting composed model are shown on fig. 17 and 18.

Fig. 17. Screening Application with Application Firewall Structure

Fig. 18. Screening Application with Application Firewall Behavior

4.3 Audit and Secure Logger Patterns

In the next step of our case study we would like to add auditing support to our design. We have selected the Audit Interceptor pattern (fig. 19 and 20) to centralize auditing functionality. An Audit Interceptor intercepts business tier requests and responses and creates audit events. Audit Interceptor depends on some secure logging facility without which it is impossible to guarantee the integrity of the audit trails. This is why we introduce also the Secure Logger pattern (fig. 21 and 22) which will ensure this additional requirement.

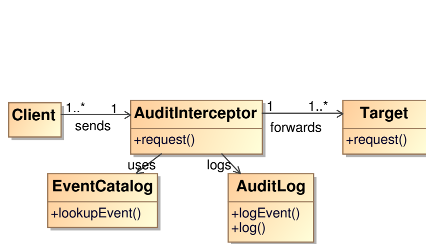

Fig. 19. Audit Interceptor Structure

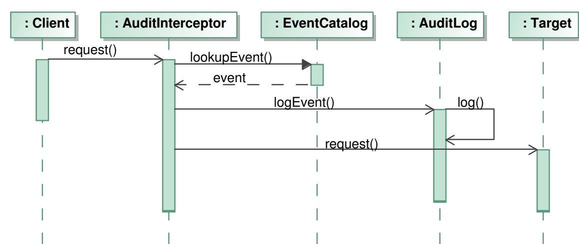

Fig. 20. Audit Interceptor Behavior

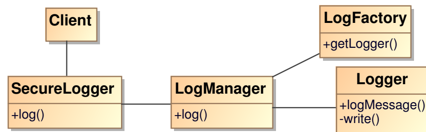

Fig. 21. Secure Logger Structure

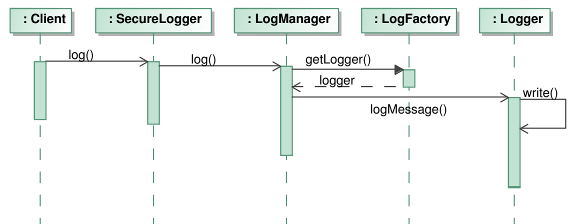

Fig. 22. Secure Logger Behavior

It is possible to apply these patterns to the application model (fig. 17) one by one. However, we chose to combine the two patterns using GReCCo, and apply the combined pattern on the application model. The *AuditLog* entity in the Audit Interceptor pattern represents the *Client* entity in the Secure Logger pattern. Hence, in order to combine the two patterns, we have to merge the two entities by relating them with a ≪*merge*≫ marked association (fig. 23). We will also place a general ordering relation between the *log* event from the *AuditLog* object and the *log* event on the *SecureLogger* (fig. 24).

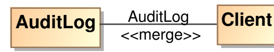

Fig. 23. Structure Composition Model

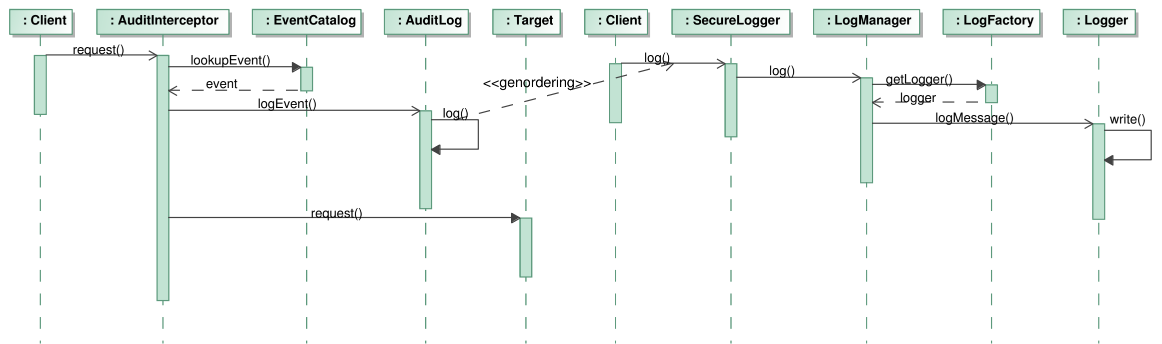

Fig. 24. Behavior Composition Model

The structure and the behavior of the combined pattern are shown on fig. 25 and 26.

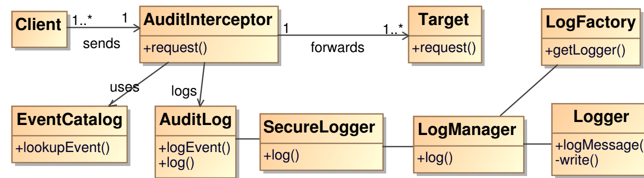

Fig. 25. Combined Structural Model

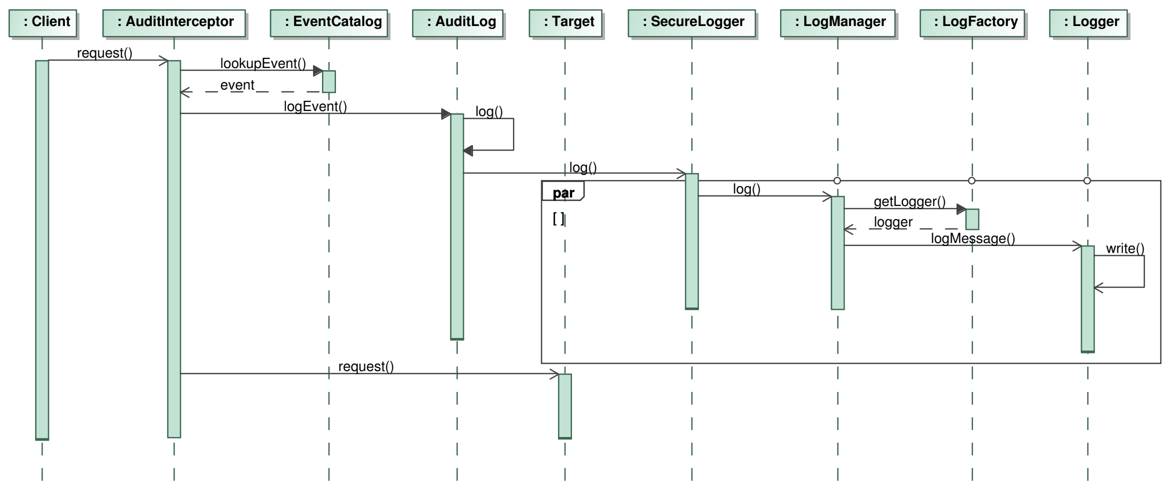

Fig. 26. Combined Behavior Model

4.4 Final Refined Application

In the final step we will compose the combined Audit and Secure Logging concerns (fig. 25 and 26) with the combined base and application firewall pattern (fig. 17 and 18). In order to realize the structural composition, we specify that *Client* and *Target* classes from the combined pattern should be instantiated by the *ApplicationFirewall* and *Server* classes from the main application respectively.

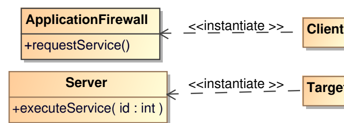

Fig. 27. Structure Composition Model

In order to compose the behavior we first notice that the auditing (using a secure logger) should be performed before the actual request is passed to the application firewall. This is why we use a loop combined fragment around the whole logging sequence and place a $\ll$ *generalordering* $\gg$ dependency to the *requestService()* event. In addition, as *request()* event is mapped to *forwardService()* event we place an additional $\ll$ *merge* $\gg$ dependency to denote that the two calls are the same (fig. 28).

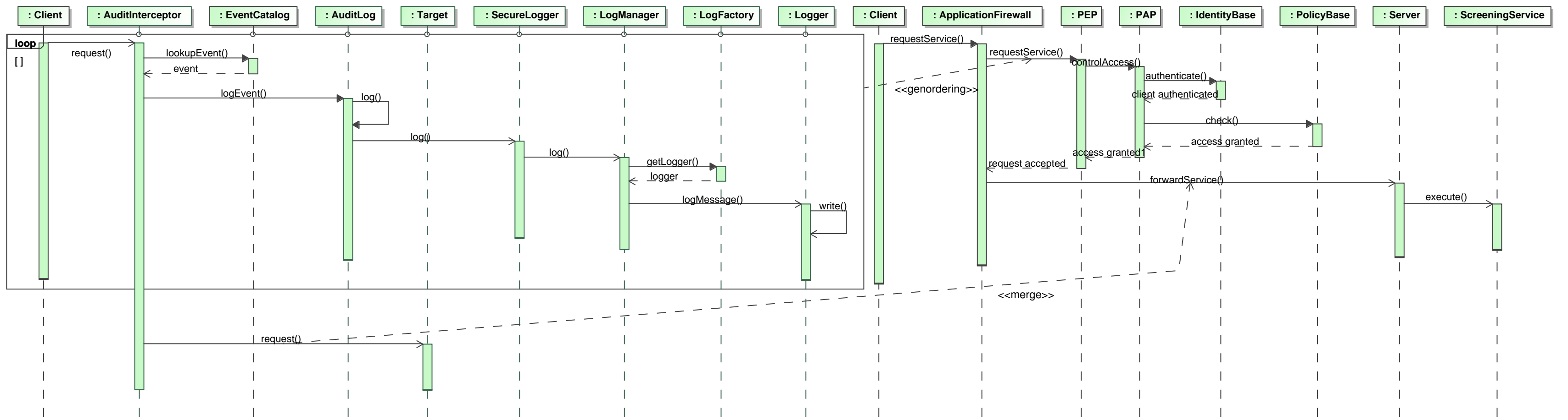

Fig. 28. Behavior Composition Model

Fig. 29 and 30 show the structure and the behavior of the final application model with Application Firewall, Audit Interceptor and Secure Logger concerns composed into it.

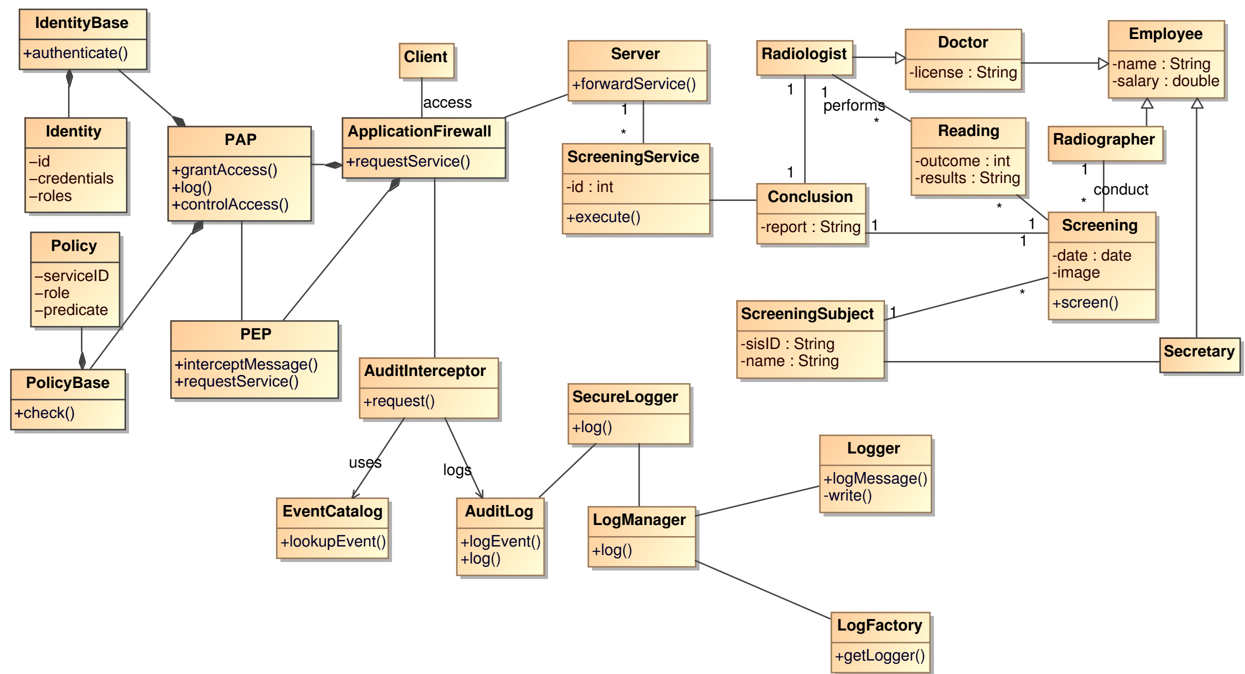

Fig. 29. Final Application Model Structure

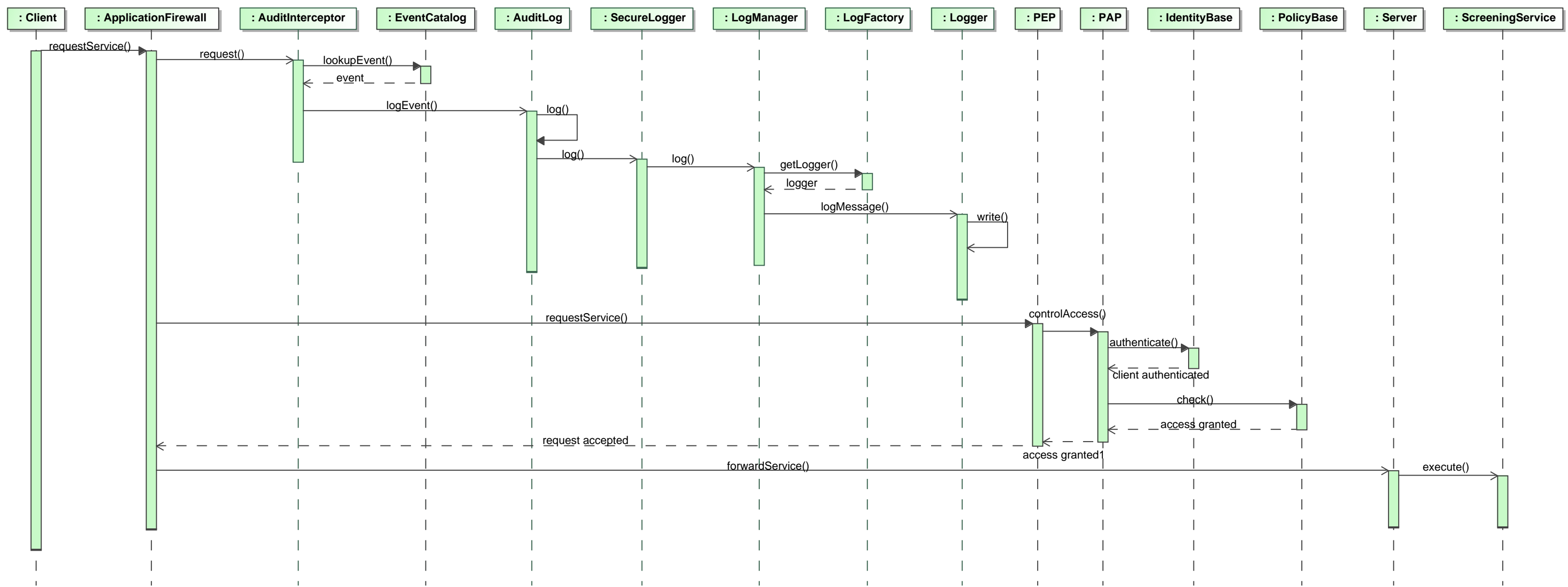

Fig. 30. Final Application Model Behavior

5 Evaluation

Our approach represents a framework for AOM using reusable concerns, whereby each concern is modeled using UML class and sequence diagrams. The approach comes with a prototype of a generic composition engine written in ATL, that can compose two given concern models. In this section, we evaluate how our approach can tackle the problems presented in section 2.

5.1 Proof-of-concept implementation

We have developed a generic engine that realizes our approach on top of ATL. For a current working version of GReCCo refer to [16]. All the proposed mapping strategies for the structural composition have been implemented and evaluated with a number of examples. The behavioral composition is not yet fully supported. In addition, the current GReCCo engine is limited to a simple prototype and has no support for traceability, preservation of manual changes during recompositions, etc.

5.2 Key requirements for reuse

We have illustrated on our case study how can we use the GReCCo approach to reuse patterns that modularize a given concern. Each of the concerns used in our case-study can be reused in another application simply by creating an appropriate composition model.

Obliviousness In our approach, each concern is modeled individually without any composition in mind. All the composition specifics, such as template parameters, join points, etc., are specified in a separate model, which is unique per composition. Given a certain concern, it is unaware of any other concerns until the composition stage. We thus achieve complete obliviousness.

Composition symmetry Our framework supports the more generic approach to concern composition - i.e., symmetrical composition. We consider all the concerns of a system as equally important. Concerns can be composed not only with the main application, but also with each other.

Interdependency management A real-life system consists usually of many concerns, each tackling a specific part of the system. These concerns are rarely completely orthogonal, but rather influence each other in one or another way. Hence, in order to have a cooachieve, concern interdependency management is a crucial point. We have reused an existing framework that helps us understand and manage the different interactions when composing different concerns.

6 Related Research

The approach of Jacobson et al. [8] represents a use case driven software development method. AOSD with use cases comes with a systematic process that focuses on the separation of concerns throughout the software development life cycle. The approach provides different means of modularization that allows one to define application-generic modules. However, Jacobson's approach does not focus on reuse, nor does it provide any guidelines on how to reuse artefacts throughout different applications and projects. Moreover, there is no tool support available.

The Theme approach of Clarke et al. [10] provides means for AOSD in the analysis phase with *Theme/Doc* and in the design phase with *Theme/UML*. A *Theme* represents a modularized view of a concern in the system. It allows a developer to model features and aspects of a system, and specify how they should be combined.

The Aspect-Oriented Architecture Models (AAM) approach of France et al. [9] presents an approach for composing aspect-oriented design class models, where each aspect model describes a feature that crosscuts elements in the primary model. Aspect and primary models are composed to obtain an integrated design view. This approach is asymmetric and allows aspect models to be composed only with the primary model.

These two approaches are generic and support reuse to some extent. However, both approaches use a template based mechanism to compose cross-cutting concerns. Each concern comes with a pre-defined set of template parameters and expects the other concern to have a certain structure or a certain functionality in order to match these. None of these approaches allows the definition and detection of concern interdependencies. Theme/UML is a symmetrical approach as any two concerns (or themes) can be composed. AAM on the other hand is asymmetric, which further limits its reusability.

Klein et al. [17,18] present an approach for specifying reusable aspect models that define structure and behavior. The approach allows expressing aspect dependencies and weaving them in a dependency-consistent manner. However, it is possible to specify and detect only one sort of relationship between aspects, namely dependency. Klein's approach is symmetric and aspects can be composed with each other. This approach uses a template-based mechanism similar to the previous two approaches for the structural composition. The behavioral composition is specified using a semantic composition algorithm. Join points are specified using a pointcut, which can be seen as a small pattern scenario. Every match of this pattern against the main scenario is a join point. As opposed to this approach we use an explicit join point selection. We believe that in most concern composition scenarios the pointcut will match against a very few and possibly one join point. In the opposite case, Klein's semantic weaving of scenarios [19] can be used in our framework for the behavioral composition.

7 Conclusions and Future Work

In this paper we have listed and discussed what we believe are key characteristics for the enhancement of concern model reuse: obliviousness, composition symmetry, concern interdependency management. We have described a new approach for specifying concerns and their compositions and illustrated on a case-study from the EHIP domain. We

have evaluated how the GReCCo approach can help us tackle each of the key qualities for improving reuse.

In the future we plan to build a library of concerns and evaluate the approach on a greater scale. Moreover, we are planning to add traceability support to our approach.

References

1. F. Jouault, I. Kurtev: Transforming models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS, Montego Bay, Jamaica (2005)
2. F. Sanen, E. Truyen, W. Joosen, A. Jackson, A. Nedos, S. Clarke, N. Loughran, A. Rashid: Classifying and documenting aspect interactions. In: Fifth AOSD Workshop on ACP4IS. (2006)
3. S. Op de beeck, E. Truyen, N. Boucké, F. Sanen, M. Bynens, W. Joosen: A study of aspect-oriented design approaches. In: Technical Report CW435, Department of Computer Science, Katholieke Universiteit Leuven. (2006)
4. D. Stein, S. Hannenberg, R. Unland: Designing aspect-oriented crosscutting in UML. In: Proc. of the 1st Workshop on AOM with UML. (2002)
5. R. Pawlak, L. Seinturier, L. Duchien, L. Martelli, F. Legond-Aubry, G. Florin: Aspect-oriented software development with java aspect components. In: Aspect-Oriented Software Development. (2005) 343–369
6. L. Fuentes, P. Sánchez: Execution of aspect oriented uml models. In: ECMDA-FA. (2007)
7. T. Cottenier, A. van den Berg, T. Elrad: The motorola WEAVR: Model weaving in a large industrial context. In: Proc. of the 6th Int. Conf. on AOSD, Vancouver, Canada. (2007)
8. I. Jacobson, P-W. Ng: Aspect-Oriented Software Development with Use Cases. Addison-Wesley Professional (2004)
9. Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, G. Georg: Directives for composing aspect-oriented design class models. In: LNCS 3880, p 75-105, Springer-Verlag. (2006)
10. E. Baniassad, S. Clarke: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley (2005)
11. F. Sanen, E. Truyen, W. Joosen: Managing concern interactions in middleware. In: Distributed Applications and Interoperable Systems. (2007) 267–283
12. E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional; 1st edition (1995)
13. K. Yskout, T. Heyman, R. Scandariato, W. Joosen: (A system of security patterns (technical report 04/12/06))
14. OMG: Uml superstructure, v2.0. OMG Document number formal/05-07-04 (2005)
15. The Protégé Ontology Editor and Knowledge Acquisition System: (http://protege.stanford.edu)
16. Generic Reusable Aspect Composition engine: (http://www.cs.kuleuven.be/~aram/implementation.html)
17. J. Klein, F. Fleureyand, J.-M. Jézéquel: Weaving multiple aspects in sequence diagrams. In: Transactions on Aspect-Oriented Software Development III. (2007)
18. J. Klein, J. Kienzle: Reusable aspect models. In: Proc. of the 11th Int. Workshop on AOM. (2007)
19. J. Klein, L. Hélouët, J.-M. Jézéquel: Semantic-based weaving of scenarios. In: AOSD. (2006)