

Composing Application Models and Security Models: on the Value of Aspect-Oriented Technologies*

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

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

Abstract. The increasing complexity and size of software applications requires improved development techniques. The introduction of aspect-oriented software development (AOSD) and the support for model-driven development (MDD) are two important and promising evolutions in this context. In this paper, we report on our exploration to identify and evaluate the synergies between both trends.

We have created a domain specific model that supports the description of an essential part of the security concern - i.e. access control. We have prototyped a model transformer that composes the security concern with an existing object-oriented application. We have developed solutions that generate source code for two different platforms: plain (OO) Java and (AO) CaesarJ respectively. We argue that targeting for AO platforms may still offer some advantages over OO platforms, even though we could weave our aspects at the design/modeling level.

Keywords: aspect oriented modeling, model weaving, model-driven development, UML 2, AOM case study

1 Introduction

Aspect-oriented software development (AOSD) is a development paradigm that enables the creation of a modular architecture for an application and its implementation. AOSD aims for a solution to the problem that many software functionalities (e.g., security, distribution, synchronization) cannot easily be architected and implemented. Based on traditional implementation techniques, the implementation of the above mentioned functionalities will be spread over the modular structure of the software application. These functionalities are said to cross-cut the modules of the application [4].

* The described work is part of the EUREKA-ITEA MARTES project, and is partly funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders).

Model-Driven Development (MDD) is a software development approach that advocates the use of models as the main software artifacts. MDD aims to deal with software complexity by offering means to specify software on different abstraction levels using modeling languages at the level of, e.g., requirements analysis, architecture, design and implementation. Transformation techniques can be applied to facilitate the generation of models at a lower level of abstraction (adding detail), as well as the merging of models (composing models). For instance, (semi-)automatic source code generation for multiple technology platforms becomes feasible.

Aspect-oriented modeling (AOM) combines ideas from AOSD and MDD by modeling parts of a complete solution and bringing them back together. This can lead to the advantage that one can use the optimal domain-specific (modeling) language for each of the concerns involved. Merging the separately designed concern models can be based on either one of two alternative approaches:

- The concern models can be translated towards a general purpose aspect-oriented platform where the platform merges the models.
- The concern models can be merged at the model level using MDD techniques. The generation of platform specific models (code) can start from the integrated model.

One may wonder whether it is better to generate code for state-of-the-art object-oriented platforms, or code for a general-purpose aspect-oriented platform. We will discuss this matter further in the paper.

In this paper we study and evaluate whether state-of-the-art MDD technologies can enable AOM as sketched above. By MDD technologies we mean the set of available standards (UML, XML, MOFScript, etc.) and tools (MagicDraw, Eclipse, etc.). We have performed a realistic case study in which we have extended a domain specific model and aspect-language for an essential part of security namely, access control. We have prototyped a model transformer in MOFScript that composes the security model with an existing application model. The key questions that have been addressed are:

- Can practical MDD technologies enable the composition of models (concerns) at a higher level of abstraction?
- What kind of production code can be generated, and what is the value of AOP code in this context?
- What are the main challenges to make this kind of approach more practical?

The paper is structured as follows. In section 2, we will briefly present earlier work on context-based access control. We will describe an overseeable case study that further motivates this work in section 3. A model-driven approach and transformations will be presented in section 4. We evaluate the results in the light of the three key questions in section 5 and we present the related work in section 6. Finally, we conclude and outline our future work.

2 Security Background

Many real world applications need to obey an access control policy in order to be deployed in practice. In general terms, such a policy would provide rules that express under which circumstances a subject is allowed to perform certain actions on the resources that are managed by the application. In practice, realistic policy rules typically rely on extra context information from the application state or the application environment. Obviously, such a policy needs to be modeled and expressed using an expressive language which enables taking into account application state. Moreover in the context of a large organization often one single organization-wide policy needs to be enforced uniformly in various applications that are deployed within the organization. Although the expression of security policies is a challenge on its own, practical and more or less de facto standard languages exist. It is actually relatively straightforward to specify basic policies using a domain-specific security policy language, such as XACML [2]. However, this approach suffers from several drawbacks. The delimitation of the responsibilities between the security officer, who manages the policy centrally, and the deployment manager of a specific application, who applies the access control policy by enforcing it to a specific application, is poorly supported. As each application within an organization may use different concepts, the same security policy may need to be specified for each application individually. The approach will render it hard to manage the policy centrally, especially if this policy is subject to frequent changes.

Verhanneman et al [9] propose a solution which tackles some of these drawbacks. Their approach allows the specification of expressive context-based policies which can be uniformly enforced in each application. The solution introduces two concepts that improve the support for the separation of concerns principle. An **access interface** abstracts from application-specific details by including only information that is relevant for access control. It consists of a number of *object interfaces* which declare the available resources and a number of *subject interfaces* which declare the available subject roles. Both interfaces declare information that the policy may need, in the form of a number of attribute declarations. In addition, an object interface declares semantic actions. These semantic actions abstract syntactic actions and represent the security sensitive operations to which policies apply.

The deployment manager, who provides the application logic, should bind the access interface to each application by means of an application-specific **view connector**. This view connector

- specifies the mapping between application objects and security objects or subjects,
- determines how to compute the necessary attributes (context information) related to the relevant security objects and subjects, and
- identifies all security sensitive operations, and maps them to corresponding semantic actions in the object interface.

The approach does not put any constraints on the choice of the authorization engine. Thus, the access control aspect can be constructed by aggregating three types of entities: a third-party authorization engine, an access interface, and the view connectors. Recall that each view connector binds the access interface to a particular application (see Fig. 1).

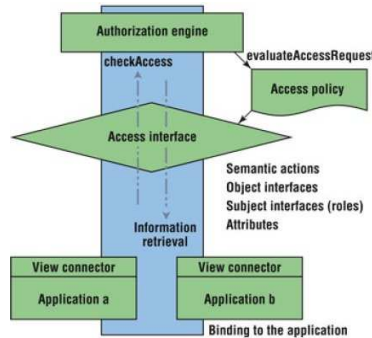


Fig. 1. Access control service

The view connector triggers an access control check each time an application event occurs that is annotated as a semantic action in the access interface. Conversely, the authorization engine can query the view connector through the access interface to retrieve required object and subject attributes. The authorization engine then evaluates whether an access request conforms to the policy.

In the work published by Verhanneman a prototype implementation of this approach has been developed for the CaesarJ platform [3]. The implementation uses a custom policy language and authorization engine and is fairly complex and error-prone. Therefore, instead of a pure AOP implementation, an AOM approach is desired to master this inherent complexity. In addition, models will provide a certain platform-independence and better code quality.

In the next section, we present a case study, which illustrates the concepts described in this section.

3 A Simple Case Study

This section presents a calendar application, and illustrates the concepts of an access interface and view connector. The calendar system lets users book appointment entries (Fig. 2). Entries can be singular or repetitive. A calendar entry involves participants and resources (rooms, projectors, etc.). The calendar system has three main actors: calendar owner, secretary and employee. The UML diagram in Fig. 3 shows the access interface for the calendar system.

Recall that an access interface provides an abstraction layer that reflects only information relevant for access control. We introduce *Calendar* and *Resource*

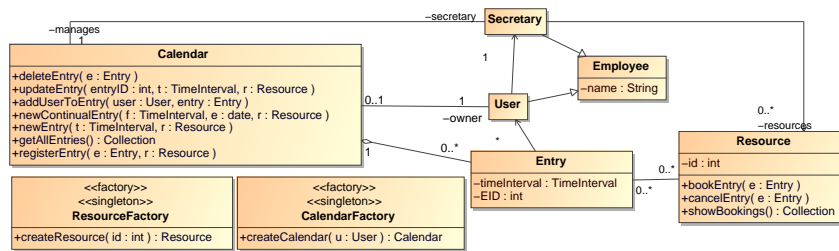


Fig. 2. Calendar Application

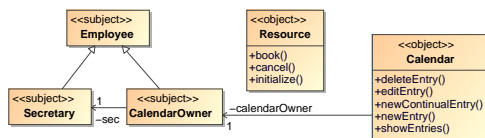


Fig. 3. Access Interface

as security objects having the set of all possible actions that security subjects can invoke on them. *Employee*, *Secretary* and *CalendarOwner* are the relevant security subjects. The security attributes which provide context information are *calendarOwner* and *sec*, which are implicitly present as navigable associations.

In order to create a mapping between these two concerns we specify the view connector a snippet of which is shown on fig. 4.

resource mapping	application.Calendar	accessinterface.CalendarObject
action mapping	getAllEntries	showEntries
	newContinualEntry	newContinualEntry

attribute computation	Calendar.getOwner()	calendarOwner

Fig. 4. View Connector

A calendar owner is an employee who can create, edit and delete singular and repetitive entries. A calendar owner’s secretary can create only singular entries and edit existing entries in the owner’s calendar. All employees can access the information in a calendar, but are not allowed to introduce any changes. Fig. 5 shows the first part of the last rule expressed in XACML.

In the next section we will show how we can use MDD techniques to bring this approach to a modeling level. Source code is then generated automatically from the specified models.

```

<Target>
<Subjects> <AnySubject/> </Subjects>
<Resources><Resource>
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Calendar</AttributeValue>
<ResourceAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"/>
</ResourceMatch>
</Resource></Resources>
<Actions><Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">showEntries</AttributeValue>
<ActionAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"/>
</ActionMatch>
</Action></Actions>
</Target>

```

Fig. 5. Access control policy

4 Model-Driven Approach

This section defines an MDD process which has been used to develop the case study presented in the previous section. Also, we shortly discuss the CaesarJ and Java generation engines.

4.1 Modeling

The first step to bring the Access Interface/View Connector approach towards an MDD process is by choosing a suitable framework representing all the necessary information on a modeling level. We will shortly discuss how each concept of the solution is modeled.

Application Business Logic Aspect Clearly the best way to represent the application logic is by using UML2 and known modeling techniques. This has already been done in the previous section (see Fig. 2).

Access Control Aspect The access interface, a third party authorization engine and a set of access control policies represents the access control aspect of the calendar system. We believe that the UML2 representation of the access interface from figure 3 is suitable for the MDD framework. We have chosen to use the XACML policy language along with Sun's implementation of the XACML authorization engine [6].

View Connector Given the two aspects of a system, we would like to express them in separate models and weave them afterwards. We can leverage on the concept of a view connector, however we need a systematic approach to represent it on a modeling level.

We represent the view controller as a non-self-contained model that references the application and access interface models using UML dependencies. We have

created a simple UML profile in order to assign different semantic meanings to the dependencies by stereotyping them. The specification of a view controller consists of three parts, which we discuss separately.

1. *Map application objects to security objects and subjects:*
Mapping application objects to security objects and subjects is done by using UML dependencies marked with the `<<map_object>>` and `<<map_subject>>` stereotypes respectively.
2. *Map operations to semantic actions:*
Mapping of operations to semantic actions is realized by UML dependencies marked with the `<<map_action>>` stereotype.
3. *Determine attribute computation for security objects and subjects:*
We consider the attribute computation as a mapping as well and represent it by using UML dependency marked with the `<<map_attribute>>` stereotype. In case a more complex computation is necessary, the approach requires that the computation is encapsulated within a helper method, and the security attribute is mapped to that method.

This set of mappings is less expressive compared to a programming language such as CaesarJ. However, we believe that in most cases this set will be sufficient. We have performed these three mapping steps on the calendar system (see Fig. 6). As the case study is relatively simple, most of the mappings are obvious. Notice that more than one concept from the application model can map to a single concept from the access interface (e.g. both *Entity* and *Calendar* classes map to the same *Calendar* object). The dependencies resulting from the third step are *calendarOwner* mapping to *owner*, and *sec* mapping to *secretary*. Some of the dependencies and stereotypes are not shown for readability purposes.

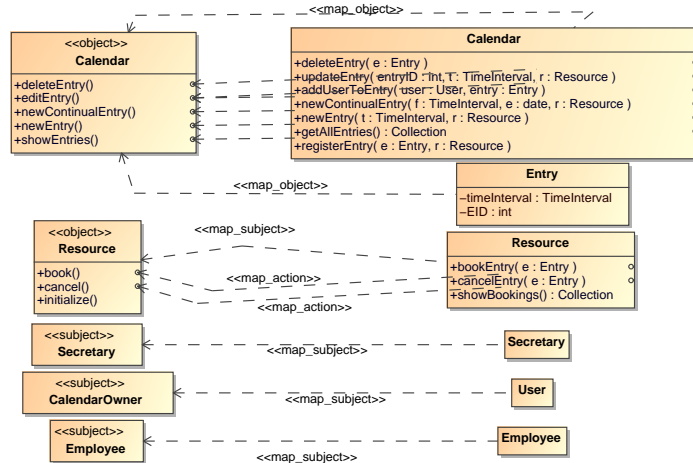


Fig. 6. View Connector

4.2 Transformation and Code Generation

Given the UML2 models of the two aspects and their binding information, we have developed model-to-code transformers for two target platforms: CaesarJ and Java. Both transformers are implemented in MOFScript [5], which takes as an input the UML2 models using the Eclipse Modeling Framework. Given the input models with a defined semantics it is relatively straightforward to write a transformer. Currently only class models are used in the transformers. Behavioral models should be implemented manually in the source.

CaesarJ Transformer The implementation of the CaesarJ transformer is straightforward. The application model is transformed to pure Java source code. The access interface and the view connector are transformed to CaesarJ specific source. The CaesarJ platform weaves the three concerns at byte-code level. We have used the around advice construct that in case of a successful authorization the original method proceeds with its execution; otherwise an exception is thrown. A more detailed description of the CaesarJ code can be found in [8].

Java Transformer Notice that the input UML2 models implicitly contain information about all the pointcuts and advice. Dependencies marked with the `<<map_action>>` stereotype denote the mapping of operations to semantic actions. These are precisely the places where the execution of the application should be stopped and directed to the authorization engine in order to perform an access control. Hence we can also perform the combination on a modeling level by injecting the access control checks just before the execution of a security specific operation using an if-then-else construct. In theory we can produce the combined woven model before generating Java code. However in practice, we skip this step, since it is easier to generate Java directly from the input models.

5 Evaluation

We have shown in this small case study that existing state-of-the-art MDD technologies can enable the composition of a primary application model with a domain-specific model of access control. The presented MDD approach is currently limited to solely addressing the security (sub)concern.

We have developed two model transformers that generate code for aspect-oriented and for object-oriented platforms. The CaesarJ transformer generates source code identical to the prototype implementation provided in [8]. The behavior of the application core should be manually enriched directly in the source code. The Java transformer generates source code equivalent to that of the CaesarJ platform. All security specific concerns are fully automatically generated. The behavior of the application model should be manually enriched directly in the source code for both transformers.

Based on our case study we have obtained equivalent source code. We have observed that the CaesarJ platform is still somewhat beneficial over the Java platform. The transformer for the AO platform is much more simple as it

transforms each of the concerns to a separate module, and leaves the weaving to the platform. Given the structural models we only generate skeleton code and leave the implementation of the behavior to the programmer. An AO platform separates the application core from the security concern, hence the AO source code is better structured and easier to maintain. In addition, consider the scenario when only the access interface or the application model changes. Given that AO platforms keep the concerns separated also in the code, we will need to regenerate only the source code for the modified model. However, AO platforms are relatively new and less stable. Moreover, the source contains control flow switches at the pointcuts to the advices which perform the access control.

The Java platform is more established and Java source code provides a better view of the thread of execution. However the Java transformator has to perform the weaving of the models, which makes the transformator more complex. The generated woven code is less structured and harder to maintain. In case only the access interface or the application core changes, we will need to regenerate and recompile the whole source code.

One of the main challenges of our approach is scalability: aspects such as reliability and distribution will need domain specific models. For each of those disciplines, one needs domain specific expertise that provides techniques and concepts similar to the access interface/view connector solution. One also needs a dedicated MDD transformator for each aspect and platform. In this context, an AO platform will be even more beneficial as its transformator will be simpler and faster to write. Finally, we did not yet explore the potential of dynamic weaving of the CaesarJ platform.

6 Related Work

Cottenier et al. [7] define an approach that weaves behavioral aspect models with the core of the system. Crosscutting behavior is designed with plain SDL statecharts from a perspective that is defined by lightweight extensions to the SDL and UML metamodels. A connector metamodel defines the structure of the aspect-to-core binding definition. Finally, a weaver behavioral metamodel defines composition primitives for specifying weaving strategies. The work defines a translationistic approach, which advocates a precise modeling of all aspects of the application including the technical implementation details. Our framework belongs to an elaborationistic approach, where only static structure is modeled and some code is generated. The rest of the code is manually enriched.

Reddy et al. [10] present 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.

Clarke et al. [1] proposes a UML design model that encompasses different separation-of-concern techniques. 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.

In contrast to these works, our approach is less generic as it encompasses only one aspect of a system. Moreover, we do not explicitly model the weaving strategy, but embed it in the transformator. Our approach defines an elaborated process though, which allows starting from high-level requirements (policies) to obtain a semi-automatically generated source code framework.

7 Conclusions and Future Work

In this paper we have evaluated an approach to apply state-of-the-art MDD technologies in the context of AOM. We have studied an AO solution that modularizes the enforcement of application-specific access control policies. We have determined that MDD technologies can enable the given solution. However, we are aware that the approach can only scale to other aspects if the related models can be managed by a lightweight transformation. In addition, we have argued that even though it is possible to weave the aspects on a modeling level we believe that AO platforms still offer certain advantages over the OO platforms.

In the future, we plan to provide metrics for a true comparison of the AO and OO solutions. Moreover we will investigate how the dynamic weaving properties of AO platforms can contribute. In addition, we will investigate the scalability of our approach, realizing other concerns.

References

1. E. Baniassad, S. Clarke. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.
2. Organization for the Advancement of Structured Information Standards. Core specification: Extensible access control markup language (XACML) v2.0. Misc.
3. M. Mezini K. Ostermann I. Aracic, V. Gasiunas. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, pages 135–173, 2006.
4. R. E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
5. SINTEF. Mofscript. <http://modelbased.net/mofscript/>.
6. SUN. XACML implementation. <http://sunxacml.sourceforge.net/>.
7. T. Elrad T. Cottenier, A. van den Berg. Modeling aspect-oriented compositions. In *Proceedings of the Satellite Events at the 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica*, 2005.
8. T. Verhanneman, F. Piessens, B. De Win, E. Truyen, W. Joosen. A modular access control service for supporting application-specific policies. *IEEE DSO*, 2006.
9. T. Verhanneman, F. Piessens, B. De Win, W. Joosen. Uniform application-level access control enforcement of organizationwide policies. In *21st Annual Computer Security Applications Conference*, 2005.
10. 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.