# An Aspect-Oriented Model Transformation to Weave Security using CVL

Jose-Miguel Horcas, Mónica Pinto and Lidia Fuentes
*CAOSD Group, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, Málaga, Spain*

Keywords: Aspect-Orientation, ATL, CVL, Model Transformations, Security, Variability, Weaving Pattern.

Abstract: In this paper, we combine the Common Variability Language (CVL) and the ATL Transformation Language to customize and incorporate a generic security model into any application that requires security. Security spans a large set of concerns such as integrity, encryption or authentication, among others, and each concern needs to be incorporated into the base application in a different way and at different points of the application. We propose a set of weaving patterns using model transformations in ATL to automatically weave the security concerns with the base application in an aspect-oriented way. Since different applications require different security requirements, the security model needs to be customized before its incorporation into the application. We resolve the variability of the security properties and implement the weaving process in CVL. We use an e-voting case study to illustrate our proposal using the CVL approach.

## 1 INTRODUCTION

In Component-Based Software Engineering (CBSE), there are properties of an application that can be dispersed and replicated in several modules. Security is an example of these properties, which are usually defined in multiple different components crosscutting the base functionality of the application. For instance, access control is defined in each component that needs to control the user rights to use a resource. An Aspect-Oriented (AO) approach aims to achieve separation of crosscutting concerns and addresses the limitation of the traditional software technologies (e.g. CBSE, Object-Oriented Programming) to appropriately modularize crosscutting concerns at the different development stages. From the perspective of AO, security is a crosscutting concern the behavior of which is tangled and/or scattered with the core behavior of the application being affected by it.

Modeling security separately from the affected application has many advantages: high reusability, low coupled components, high cohesive software architectures. To benefit from these advantages security requirements need to be taken into account from early stages in the development process — i.e. at the architectural level. Moreover, the software architecture modeling the security functionality should be defined separately from the software architecture of the base applications that need it. Separating security related concerns from the application base code is also the main motivation of the INTER-TRUST project[1] that is under development. With this project, the industrial partners demand security solutions easily instantiable as part of any application. The approach presented in this paper pursues an answer to these demands. However, security spans a large set of concerns, including encryption, authentication, access control, and authorization, among others, and each of these concerns affects the base application in a different way. For instance, access control is performed before the execution of a restricted action by the user, while encryption is performed before sending a message through a network in order to encrypt the information, but also after receiving the message in the target in order to decrypt the information. Moreover, not all the applications require all the security concerns. A first application may require the integrity and the non-repudiation concerns, a second application may require only the encryption concern, and a third application may also require the encryption concern but using a different encryption algorithm. The software architecture of these applications should include only the necessary security functionality and the concerns that are not required should not be part of the final architecture of the application.

In this paper, we follow an Aspect-Oriented Modeling (AOM) approach[2] to incorporate (*weave* in the AO terminology) a customized security model into a

[1] http://www.inter-trust.eu//

[2] http://www.aspect-modeling.org

Horcas J., Pinto M. and Fuentes L..
An Aspect-Oriented Model Transformation to Weave Security using CVL.
DOI: 10.5220/0004890601380147
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 138-147
ISBN: 978-989-758-007-9

base application that has been specified independently — i.e. the application does not contain any security concern and the security model has been defined generically in order to reuse it in several applications. This is done automatically without manually modifying the existing elements in the model of the base application. To do this, we use the Common Variability Language (CVL) (Haugen et al., 2012) in combination with the ATL Transformation Language (Jouault et al., 2008). CVL allows us to specify and resolve the variability of the security model, and also allows us to weave the customized security model with the base application using model transformation rules, automatically generating the complete model of the application with the security functionality. CVL includes the possibility of delegating its control during variability resolution to a Model-2-Model (M2M) transformation engine such as ATL, QVT (Query/View/-Transformation), etc. The main contribution of this paper is that we define a set of reusable weaving patterns in CVL to incorporate each security concern in the most suitable place (*join point*) of the base application model. We define the semantic of each weaving pattern using reusable ATL transformation rules that are different for each security concern since each of them need to be woven with the base application in a different way.

The advantage of using CVL is that it allows us to define the models in any language based on Meta-Object Facility (MOF) meta-models. In this paper we use the Unified Modeling Language (UML) to define the models (software architectures as component diagrams) and the weaving patterns, but our proposal is suitable for use with any MOF compliant language, and the weaving patterns are reusable by defining a previous model transformation between UML and the language used to define the application and the security software architectures. In addition, the security software architecture can also be reused with any other application of the same domain by reusing the model transformations.

In contrast to other variability techniques used by traditional Software Product Lines (SPLs), such as feature models that require an additional process to generate the customized software architecture from the feature model configuration, CVL is intended to be used in conjunction with architectural models, resolving the variability and generating the architectural configuration in the same process. Furthermore, CVL was submitted to the Object Management Group (OMG) as standard to model variability.

The rest of this paper is structured as follows. In Section 2 we present the case study used throughout the paper. Section 3 introduces our proposal using CVL and briefly describes the CVL terminology. Section 4 explains how we perform the configuration of the security model and the weaving process. In Section 5 we provide the weaving patterns for the security concerns through model transformations. Section 6 surveys related work and in Section 7 we conclude the paper and consider future directions.

## 2 CASE STUDY

Our case study is an electronic voting (e-voting) application which is one of the demonstrators of the INTER-TRUST project. E-Voting is one of the environments where security requirements are complex. Figure 1 shows a simplified software architecture in UML with the main functionality of an e-voting application. This architecture does not include any component related to the security requirements. The Voter Application component allows clients to cast their votes from smart phones, tablets, e-mails, etc. by using the EVotingInt interface. The Vote Server component receives the votes and the Election Data stores them in a digital ballot box through the VoteStorageInt interface. Administrators can manage the election data and get the election results through the VotingMngInt interface that provides access to the functionality of the Election Data and the Vote Counting components.

Apart from the base functionality shown in Figure 1, the e-voting application requires a list of security extra-functional properties. Concretely, it is of paramount importance to guarantee that: (1) all the votes in the digital ballot box belong to an eligible voter (i.e. integrity of the votes); (2) at the same time the privacy of the voter must be preserved, even in the counting process (i.e. votes must be protected by means of cryptography); (3) the voter must be authenticated using a personal digital certificate, such an electronic ID card, and (4) administrators must be authorized to perform actions over the election data.

With the goal of defining the security functionalities once, and reusing them for several applications, Figure 2 shows a UML software architecture with the complete functionality of all the possible security concerns. This includes the Integrity, Authentication, Encryption, Authorization, and Digital Signature components with all kinds of authentication mechanisms, encryption algorithms, and the integrity, authorization, and digital signature functionality.[3] However,

[3] To simplify the case study we do not show all the existing security properties nor all the existing algorithms for each concern.

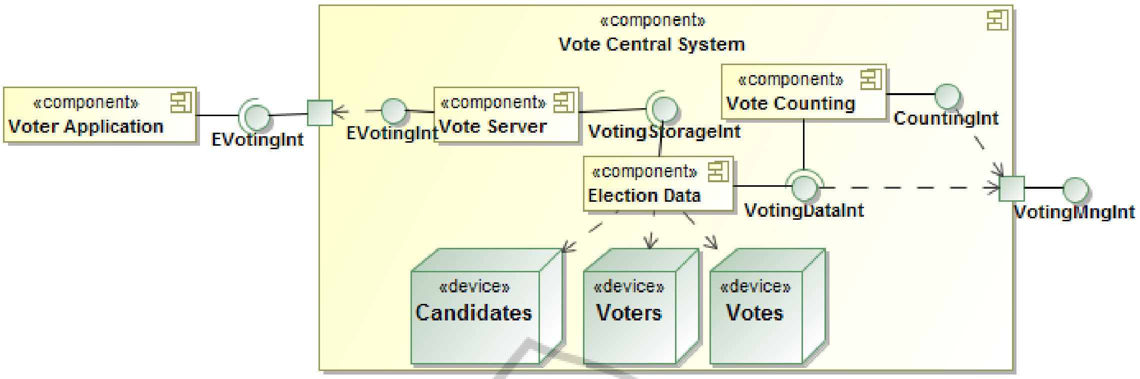

Figure 1: e-Voting software architecture.

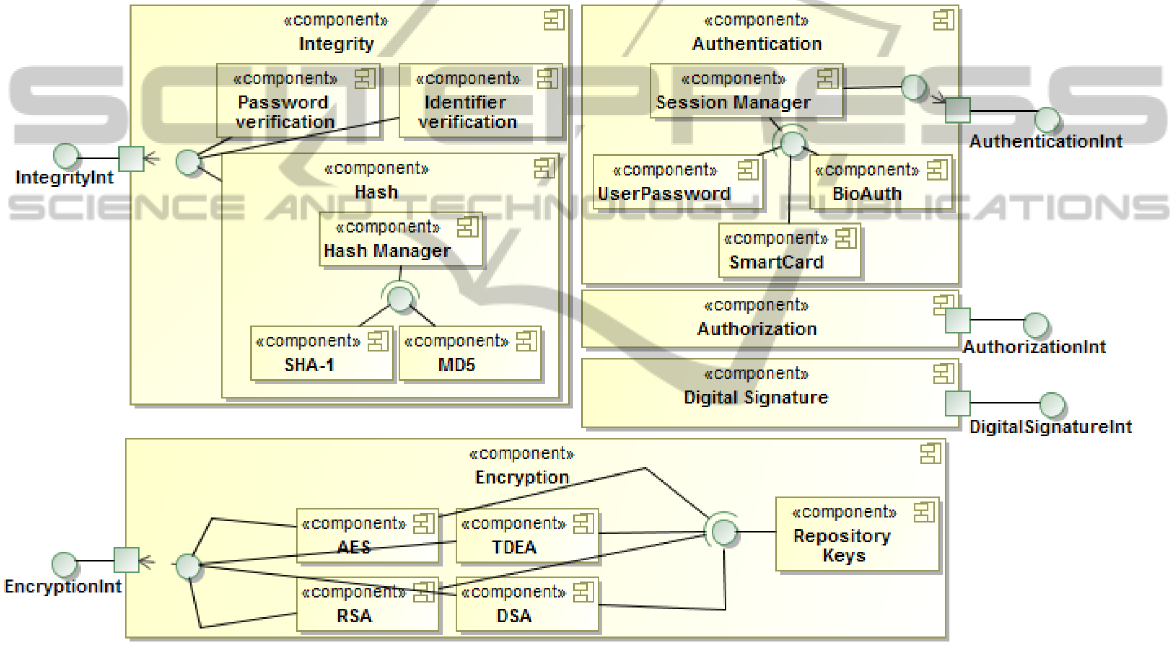

Figure 2: Security software architecture.

the e-voting application only needs a particular configuration of these security functionalities based on the previous security requirements.

## 3 OUR PROPOSAL USING CVL

CVL is a domain-independent language for specifying and resolving variability. It makes the specification and resolution of variability over any instance of models defined using a MOF-based meta-model easier.

Figure 3 shows our proposal using the CVL approach. The core software architecture of our base application and the security software architecture with all the security functionalities are the Base Models and can be defined in any MOF-defined language. The specification of the variability for the security concerns is expressed in an abstract level in the Variability Model. The specification of concrete variability in the security model and the security weaving patterns for each security concern are also defined in the variability model. Different configurations of the security model are provided in the Resolution Models. These configurations are selections of a set of choices in the variability model.

CVL provides an executable engine to automatically produce the Resolved Models taking as inputs the variability model, the resolution models and the base models. In our proposal, the resolved models are the software architecture of the base application

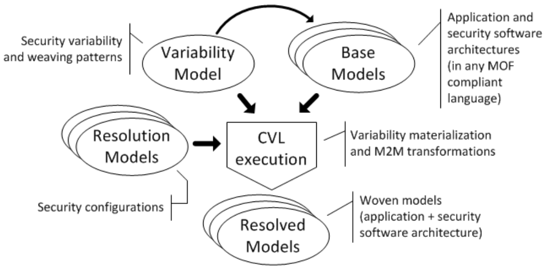
Security variability and weaving patterns
Variability Model
Base Models
Application and security software architectures (in any MOF compliant language)
Resolution Models
CVL execution
Variability materialization and M2M transformations
Security configurations
Resolved Models
Woven models (application + security software architecture)

Figure 3: Our proposal using the CVL approach.

woven with the requested security configuration. The process of deriving a resolved model from a base model given a resolution model is called *materialization*. Apart from resolving variability, the CVL engine also has the capability to delegate its control to a M2M transformation engine. This is especially useful for defining domain specific actions the semantics of which are not defined by CVL, and it is used in our proposal to implement the weaving process between the application architecture and the security configuration architecture by performing models transformations during the execution of CVL.

In order to implement our proposal we use the following concepts of CVL[4]:

**Variability Specifications (VSpecs).** They are part of the variability model. They are tree-based structures representing choices,[5] and can have *variation points* bound to them. To materialize a base model with a variability model over it, resolutions for the VSpecs must be provided. Choices are resolved by deciding them negatively or positively.

**Variation Points.** They are part of the variability model and define specific modifications to be applied to the base model during materialization. They refer to base model elements via base models handles and are bound to VSpecs. The application of the variation points depends on the resolution for the VSpecs.

**Existence Variation Point.** It is a kind of variation point that indicates the existence of a particular object, link, or value in the base model. Its negative application involves deleting elements from the base model.

**Opaque Variation Point (OVP).** It is a kind of variation point the impact of which on the base model is user-defined through a model transformation language. OVPs allow extending and customizing the semantic of the existing CVL variation points.

The AO main concepts that we use are:

**Join Point.** It is a point in the model (e.g. a method call in an interface) which can be affected by the crosscutting behavior.

**Advice.** It is the additional behavior that affects the base program at the selected join points. There are usually three kinds of advices based on 'when' the behavior takes place in reference to the join points: **before**, **after**, and **around**. Around advices allow bypassing the execution of the captured join points.

**Pointcut.** It is an expression that describes a set of join points.

## 4 SECURITY WEAVING

In order to incorporate a particular configuration of the security functionality into the base application of our case study, we implement the weaving process using CVL and, concretely, using the OVPs of CVL. Before that, we need to customize the security software architecture from the security requirements of our e-voting application. Both processes, the selection of a security configuration and the weaving are performed in the same step using CVL.

Figure 4 shows an instance of our proposal using the CVL approach with our case study and the security specifications. The variability model for security is specified in an abstract level using VSpecs (top of Figure 4). Security properties are decomposed into choices in the VSpecs, indicating which security concerns are optional and which are mandatory. In our case study, for simplicity, security is decomposed only into the choices of Integrity, Access Control which in turn contains the Authentication and Authorization concerns, and Cryptography that contains the Encryption and Digital Signature concerns. Each of them is also composed by the available methods and algorithms. For instance, there are three kinds of methods to verify the integrity of the data: Password verification, Data identifier, and Hash verification. The last one can be performed by using the MD5 or the SHA-1 algorithm. The resolution of all these choices require a yes/no decision, and the security configuration (the resolution model) is a selection of this set of choices in the VSpecs — i.e. the security concerns that are decided positively (darkened choices in Figure 4).

The concrete variability of security and the weaving patterns are specified using variation points (middle of Figure 4): "object existence" variation points to realize the variability and OVPs to do the weav-

[4] The complete description of CVL can be found in http://www.omgwiki.org/variability/.

[5] "Features" in most SPL approaches.

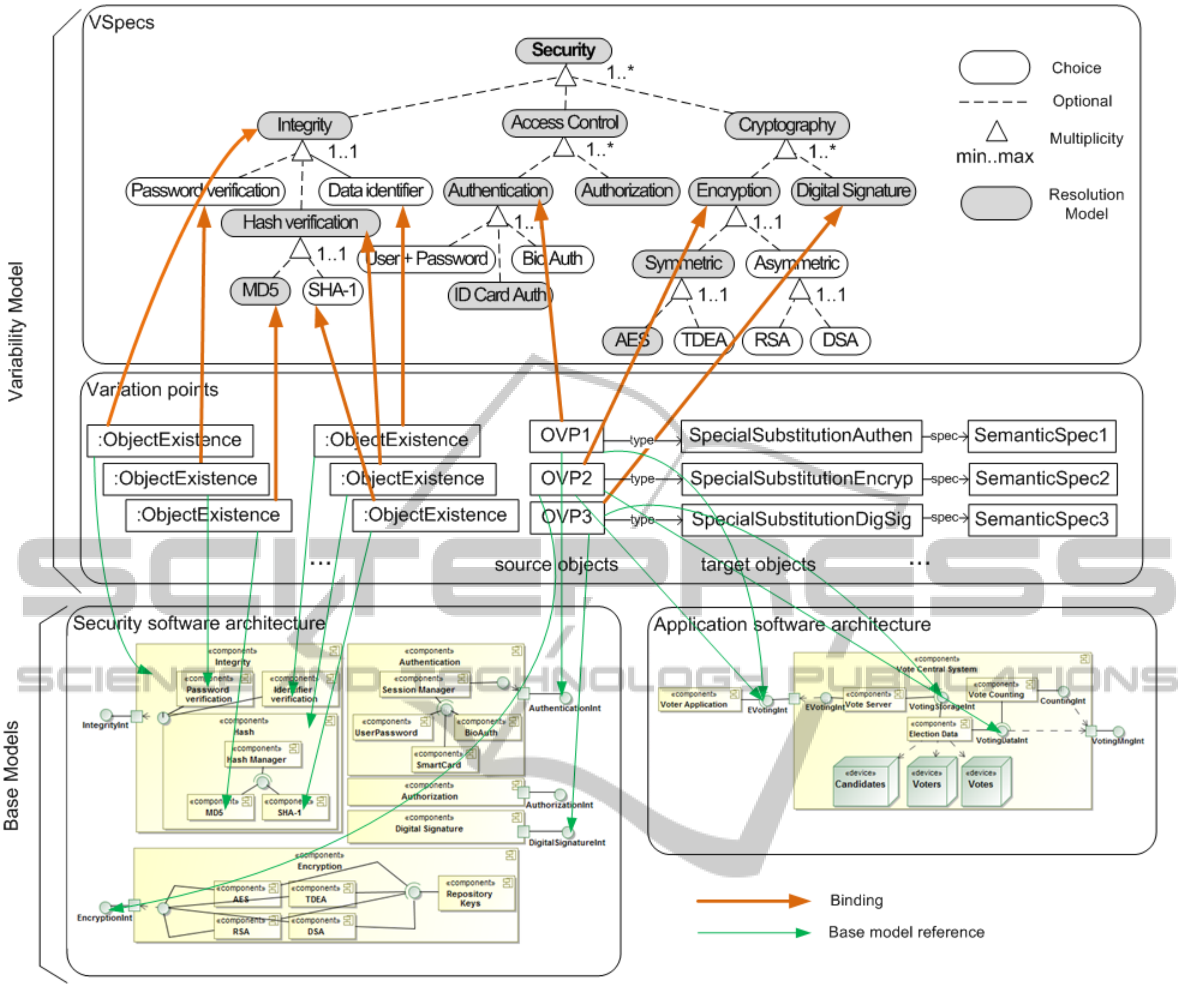

Figure 4: Security configuring and weaving using CVL.

ing. The object existence variation points are bound to choices of the VSpecs and refer to components of the security software architecture (bottom of Figure 4). This kind of variation point indicates the existence of a particular object (component) that will be included or removed from the security software architecture based on the resolution provided for the associated VSpec. For instance, the variation point bound to the Integrity concern in the VSpecs (:ObjectExistence) indicates that if the Integrity choice is decided positively (i.e. is selected in the resolution model) in a configuration, the related elements (the Integrity component and its interfaces with their attachments) in the security software architecture will exist in the resolved model and if integrity is decided negatively (i.e. is not selected in the resolution model) those related elements will be removed from the resolved model.

The OVPs are also bound to the VSpecs but have two or more references in the base models: (1) one reference (source objects) to the interface in the security software architecture the behavior of which we want to incorporate in our base application — i.e. the advice, and (2) one or more references (target objects) to the interfaces in the application software architecture where we want to incorporate the security concern — i.e. the join points. For instance, OVP2 has a reference (sourceObject) to the EncryptionInt interface in the security model and two targets (targetObjects): one for encrypting (that references to the EvotingInt interface in the application model) and one for decrypting (that references to the VotingDataInt interface).

OVPs are also bound to an OVPType, where this type explicitly defines the semantic of the special substitution — i.e. the transformation rules to weave the security concerns into the base application. Each security concern needs to be woven with the base application following a different transformation pattern based on the aspectual information of the concern: the kind of the advice that the concern implements: *before*, *after*, or *around*; the method (advice) that must be executed by the concern; and the intercepted methods (join points) in the base application. So, using CVL we need to use several variation points, with different semantics, to indicate how the elements of the models are adapted in order to generate the resolved model. During variability materialization, the CVL engine will delegate its control to a M2M transformation engine (ATL in our proposal) whenever it encounters an OVP. The M2M transformation engine executes the semantic specification associated with the OVP and resolves the variability accordingly.

The resolved model is automatically generated (Figure 5) and the security configuration is woven with our application software architecture: the components related to the security concerns are clearly visible in the static part of the architecture, and the relationships between the security elements and the elements of the base application ("crosscuts" dependency relationship) explicitly indicate that the sources of the relationships crosscut the architectural level, and the targets are the point of the application where they take place. However, the aspectual information with the interactions between the components is not represented in the software architecture of Figure 5. To complete the design we complement the software architecture with a set of sequence diagrams that represent the aspectual information and that are also automatically generated by the weaving patterns (Pinto et al., 2009). The following section shows the weaving patterns, in detail, for each security concern.

## 5 SECURITY WEAVING PATTERNS

The nature of each security concern avoids having to have a unique and homogeneous weaving pattern. As we explained in the previous section each concern needs different aspectual information and the weaving patterns (i.e. the transformation rules) needed to perform the weaving are different.

The semantic specification associated with each OVP must include transformation rules to: (1) incorporate the security components into the software architecture of the base application; (2) create the "crosscuts" relationships between the interface of the concern (the advice) and the interface of the application where the crosscuts take places (the join point); and (3) generate the sequence diagram that represents the behavior of the crosscutting relationship.

We define a set of ATL transformations for each of the security concerns: authentication, encryption, authorization, integrity, and digital signature. Weaving patterns for other security or crosscutting concerns may be defined in a similar way. To perform the weaving between the models, each ATL transformation takes as input the two models (the application model and the security model) and generates as output the same application model with the appropriate security concern merged. The elements of the application model remain unchanged in the output model. So, we can focus on the generation of the security elements in the ATL transformations.

The transformation rules (weaving patterns) are defined only once and can be reused in each application. However, there is specific information of the base application that the software architect must provide because it is different for each application. For instance, the software architect must define the concrete pointcuts (e.g. the method signature) for the join points in the base application. To simplify the case study, we only use method call/execution pointcut designators. In order to make the transformation rules more reusable, we define the aspectual information as attributes (*helpers*) in ATL that must be filled in for each application with the signature of the intercepted method by the crosscut relationship.

### 5.1 Authentication

The semantic of the special substitution for the authentication concern is shown in the Listing 1. The transformation rule: (1) copies the authentication security elements: the component (sourceComp) and interface (sourceInt) from the SecurityModel to the application model (AppModel in the rule); (2) creates the "crosscut" relationship between the source (sourceInt) and the target (targetInt) interfaces; and (3) generates the sequence diagram with the interactions between the authentication and the base application elements.

The aspectual information is coded in the transformation pattern and is used to generate the interactions. For instance, to generate the authentication sequence diagram we use a *called rule* of ATL with the aspectual information: the intercepted method vote(Object) (provided with the helper operation), the advice ('authenticate()'), and the kind of the advice ('around'), apart from the interfaces and components related.

The sequence diagram generated is shown in Figure 6. Authentication is usually performed *before* a method call, however we use an *around* advice in order to abort the call to the method vote(Object) if the authentication fails.

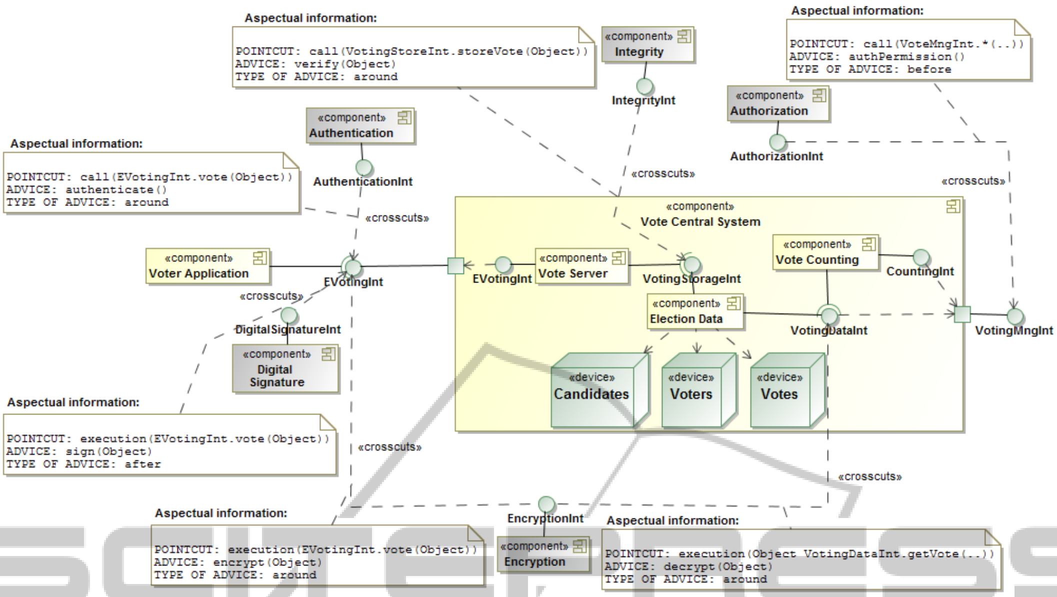

Figure 5: Application with security functionality woven.

Listing 1: SemanticSpec1 (SpecialSubstitutionAuthen)

```
module authentication;
create OUT:UML from AppModel:UML,
                    SecurityModel:UML;

rule Authentication {
 from sourceInt : UML!Interface in SecurityModel,
     sourceComp : UML!Component in SecurityModel,
     targetInt : UML!Interface in AppModel,
       (sourceInt.name = 'AuthenticationInt' and
        sourceComp.name = 'Authentication' and
        targetInt.name = thisModule.targetObject)
 to securityComp : UML!Component (...),
    securityInt : UML!Interface (...),
    crosscutAssoc : UML!Dependency(
      client <- sourceInt,
      supplier <- targetInt, ...)
 do {crosscutAssoc.applyStereotype(
      thisModule.getStereotype('crosscuts'));
     thisModule.CreateAuthInteraction(targetInt,
       sourceInt, sourceComp, 'around',
      thisModule.operation, 'authenticate()');}
}
```

## 5.2 Encryption

The semantic of the special substitution for the encryption concern (Listing 2) is different since it uses two different advices ('encrypt(Object)' and 'decrypt(Object)') in two different points of the application. The votes are encrypted in the vote(Object) method of the EVotingInt interface (provided by the operEncrypt helper), and are decrypted in the getVote() method of the VotingDataInt interface (provided by the operDecrypt helper). The transformation rule automatically generates two different sequence diagrams with that information: one for encrypting (Figure 7) and one for decrypting (Figure 8).

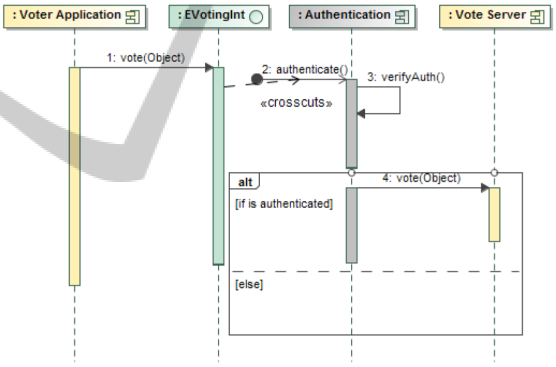

Figure 6: Authentication sequence diagram.

Listing 2: SemanticSpec2 (SpecialSubstitutionEncrypt)

```
module encryption;
create OUT:UML from AppModel:UML,
                    SecurityModel:UML;

rule Encryption {
 from sourceInt : UML!Interface in SecurityModel,
     sourceComp : UML!Component in SecurityModel,
     targetEncrypInt : UML!Interface in AppModel,
     targetDecrypInt : UML!Interface in AppModel,
       (sourceInt.name = 'EncryptionInt' and
        sourceComp.name = 'Encryption' and
        targetDecrypInt.name = thisModule.
             targetObject1 and
        targetEncryptInt.name = thisModule.
             targetObject2)
  to securityComp : UML!Component (...),
     securityInt : UML!Interface (...),
     crosscutAssoc1 : UML!Dependency(
       client <- sourceInt,
       supplier <- targetEncryptInt, ...),
     crosscutAssoc2 : UML!Dependency(
       client <- sourceInt,
       supplier <- targetDecryptInt, ...)
  do {crosscutAssoc1.applyStereotype(
       thisModule.getStereotype('crosscuts'));
      crosscutAssoc2.applyStereotype(
       thisModule.getStereotype('crosscuts'));
     thisModule.CreateEncryptInteraction(
        targetEncryptInt, sourceInt, sourceComp,
        'around', thisModule.operEncrypt,
        'encrypt(Object)');
     thisModule.CreateEncryptInteraction(
              targetDecryptInt, sourceInt,
              sourceComp, 'around',
              thisModule.operDecrypt,
              'decrypt(Object)');}
}
```

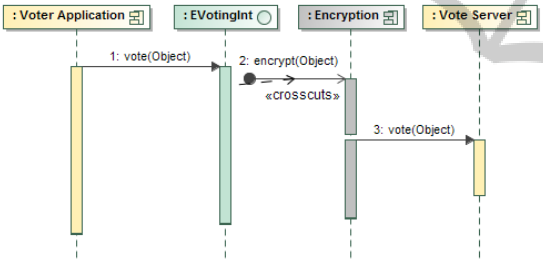

Figure 7: Encryption sequence diagram for encrypting.

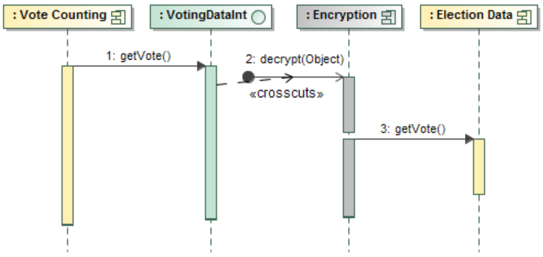

Figure 8: Encryption sequence diagram for decrypting.

## 5.3 Digital Signature

The weaving pattern for the digital signature concern is very similar to the encryption pattern, but we only need the 'sign(Object)' advice. We use an *after* advice to sign the votes in the server side after sending them in order to prevent device clients with lower resources from casting their votes. The sequence diagram of Figure 9 shows that the advice 'sign(Object)' is performed after the execution of the method vote(Object).

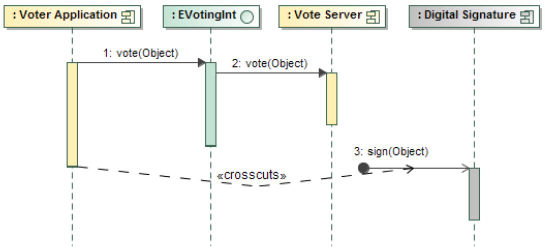

Figure 9: Digital signature sequence diagram.

## 5.4 Integrity

The integrity concern guarantees the votes belong to an eligible voter. The weaving pattern is also similar to the encryption pattern. We use an *around* advice over the method storeVote(Object) of the VotingStoreInt interface with the purpose of verifying the authenticity of the vote and rejecting it if the vote belongs to an invalid voter (see the sequence diagram of Figure 10).

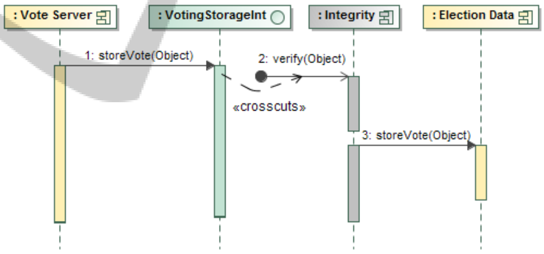

Figure 10: Integrity sequence diagram.

## 5.5 Authorization

The authorization weaving pattern is similar to the authentication case, and in most security approaches the authorization concern is based on different authentication mechanisms. In our e-voting application we use a *before* advice in order to grant or deny permissions to access privileged data of the election process. The sequence diagram (Figure 11) shows that the authorization logic verifies the permissions of the administrator before calling any method of the VotingMngInt interface.

# 6 RELATED WORK

Security is usually achieved in several ways, but most of the approaches present the security as a set of non-

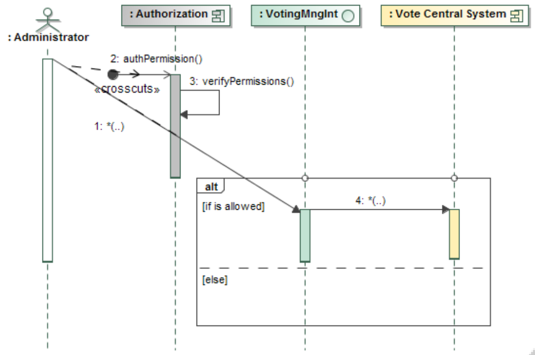

Figure 11: Authorization sequence diagram.

functional properties, instead of focusing on the functional part of the security concerns as we do. For instance, in (Georg et al., 2002), the authors analyze the impact of security properties on other functional concerns of the base application using an AO approach.

Model Driven Engineering (MDE) has also been used in the field of SPLs (Sijtema, 2010). Sijtema proposes a strategy to let ATL handle the variability by extending the concrete syntax of ATL with the concept of variability rules. Variability rules are used in the context of a transformation sequence which successively refines models. However, they first model the variability separately in a feature diagram and have to make the correspondence between the feature selections and the realization of the artefacts. In comparison with our proposal, using CVL we model the variability and bind the features directly to the elements in the software architecture. We use the basic ATL without the need to extend it, but our proposal can also be used with other transformation languages such as QVT or ETL (Epsilon Transformation Language) (Kolovos et al., 2008).

Recently, CVL has been applied in multiple approaches. For instance, CVL is used to manage the variability in the context of software processes (Rouillé et al., 2012), business process (Ayora et al., 2012), or even for synthesizing an SPL using model comparison (Zhang et al., 2011). In (Combemale et al., 2012) CVL is used to specify and resolve the variability of a software design, and the Reusable Aspect Model (RAM) technique is used to specify and compose the detailed structural and behavioral design models corresponding to the chosen variants. Our approach, in contrast, focuses on the architectural level in the development process, and we perform the weaving process with the CVL engine, instead of using an external RAM weaver.

## 7 CONCLUSIONS AND FUTURE WORK

We have defined a set of security weaving patterns through model transformations in ATL that allows the automatic incorporation of a customized security model into the base application model by using CVL and AOSD. CVL makes our proposal suitable for use with any MOF based model. We have implemented our proposal and used it with several case studies such as the e-voting application presented in this paper or in a vehicle-to-vehicle and vehicle-to-infrastructure application that is another of the demonstrators of the INTER-TRUST project. In all cases, we have used UML as the modeling language for the software architectures and we conclude that our proposal improves the modularity and reusability of both software architectures, the core architecture of the application and the security software architecture.

As part of our future work, we plan to make the transformation rules more reusable by using a third *binding model* in the weaving patterns (Durán et al., 2013). The binding model allows defining the aspectual information (e.g. pointcut definitions, advices) as external parameters to use them in the transformation rules independently from the input models. We also plan to take into account the existing dependencies between the security concerns (e.g. authentication is usually needed by the authorization concern), and how these dependencies affect the weaving patterns and the sequence diagrams when two or more concerns are applied in the same point of the application. Moreover, we plan to define weaving patterns for other crosscutting concerns such as usability, persistence, context-awareness, etc.

## ACKNOWLEDGEMENTS

Work supported by the European Project INTER-TRUST 317731 and the Spanish Projects TIN2012-34840 and FamiWare P09-TIC-5231.

## REFERENCES

Ayora, C., Torres, V., Pelechano, V., and Alférez, G. H. (2012). Applying CVL to business process variability management. In *Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone*, VARY '12, pages 26–31, New York, NY, USA. ACM.

Combemale, B., Barais, O., Alam, O., and Kienzle, J. (2012). Using CVL to Operationalize Product Line

Development with Reusable Aspect Models. In *VARY@MoDELS'12: VARiability for You*, Innsbruck, Autriche. ACM. VaryMDE (bilateral collaboration between Inria and Thales).

Durn, F., Zschaler, S., and Troya, J. (2013). On the reusable specification of non-functional properties in DSLs. In Czarnecki, K. and Hedin, G., editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 332–351. Springer Berlin Heidelberg.

Georg, G., France, R., and Ray, I. (2002). An aspect-based approach to modeling security concerns. In *Proceedings of the Workshop on Critical Systems Development with UML*, pages 107–120.

Haugen, O., Wąsowski, A., and Czarnecki, K. (2012). CVL: common variability language. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 266–267, New York, NY, USA. ACM.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1–2):31–39.

Kolovos, D., Paige, R., and Polack, F. (2008). The epsilon transformation language. In Vallecillo, A., Gray, J., and Pierantonio, A., editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg.

Pinto, M., Fuentes, L., Fernndez, L., and Valenzuela, J. (2009). Using AOSD and MDD to enhance the architectural design phase. In Meersman, R., Herrero, P., and Dillon, T., editors, *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 360–369. Springer Berlin Heidelberg.

Rouillé, E., Combemale, B., Barais, O., Touzet, D., and Jézéquel, J.-M. (2012). Leveraging CVL to Manage Variability in Software Process Lines. In *Asia-Pacific Software Engineering Conference*, Hong Kong, Chine.

Sijtema, M. (2010). Introducing variability rules in atl for managing variability in MDE-based product lines. *Proc of MtATL*, 10:39–49.

Zhang, X., Haugen, O., and Moller-Pedersen, B. (2011). Model comparison to synthesize a model-driven software product line. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 90–99.