# SUMMARY:
# MODEL DRIVEN SECURITY

JAN-FILIP ZAGALAK, JZAGALAK@STUDENT.ETHZ.CH

**Model Driven Security: From UML Models to Access Control Infrastructres**
David Basin, Juergen Doser, ETH Zuerich
Torsten lodderstedt, Interactive Objects Software GmbH Freiburg
September 2005

## 1. Model Driven Security: an Overview

Requirements affecting security are often poorly integrated in the overall system software development process. Reasons for this may be differences in the activities carried out to achieve the project goal or the kind of used representations (text vs. graphical representation). As a consequence the security engineering part, which deals with the mentioned security sensitive aspects, is often carried out decoupled and parallel to the system design engineering development process (see Figure 1).

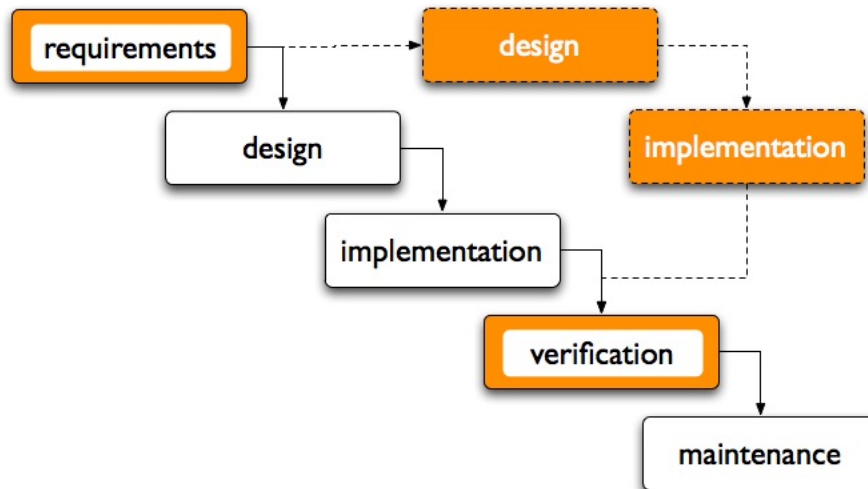


Figure 1. security requirements (colored) in a common software engineering process

This leads obviously to drawbacks and quality issues in the deployed software / system. The two separately developed parts are integrated in a late process phase. This ad hoc integration of security handicaps the full evolvement of projected security mechanisms and negatively impacts maintainability and extendability. Neither can a security

engineer, responsible for the successful realization of security, be sure that all projected security requirements were implemented properly, nor has he good confidence about the quality of the implemented security features. Again the causes are the following two gaps in a (common) software engineering development process:

- security design models and system design models are typically disjoint and expressed in different ways, lack of consequent traceability of security requirement through all development phases
- ad hoc integration of security mechanisms in late development phases

The concept of Model Driven Architecture (MDA) suggests the development of an abstract high-level model and then uses automatic transformation functions to generate target platform code e.g. EJB, .NET out of the model. As the name implies only the architecture is specified in the model, the business logic has to be defined separately from the model (see Figure 2).

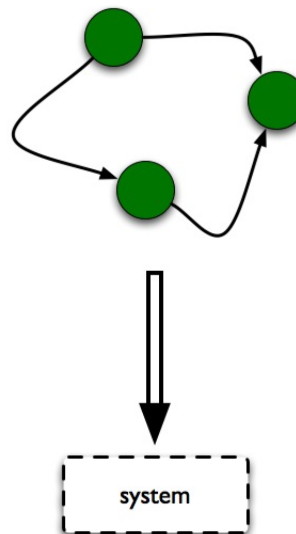


FIGURE 2. Model Driven Architecture

This paper uses the MDA concepts and extends it with security modeling abilities which becomes MDS model driven security. A MDS model contains system and security properties and therefore offers a common representation for both domains. According to MDA a model can be automatically transformed to source code through transformation functions. But as before no business logic is generated, but the architecture together with the security mechanism (see Figure 3). To provide a problem domain independent solution the authors don't present a concrete language but a general schema how to combine a system design modeling and a security modeling language to a new security design language that encapsulates both domains. This paper claims to offer a schema for every possible security concept, but only a role-based

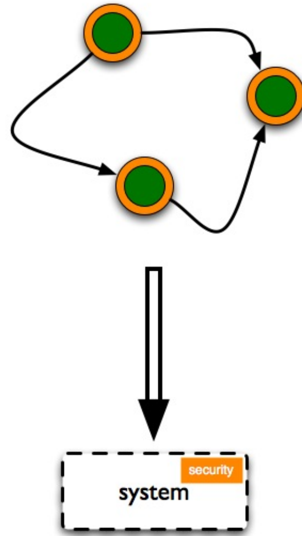


FIGURE 3. Model Driven Security

access control (RBAC) concept is used and applied to several examples.[1] The schema of building a security design language consists of tree components:

- a system design modeling language for constructing system design models
- a security modeling language for expressing security policies
- a dialect to provide a mapping between concrete protected system resources in the design modeling language and abstract security sensitive resources[2] in the security modeling language

The dialect serves as glue or bridge between the security and the system design modeling language. Problem domain specific issues are encapsulated in the dialect and keep the security modeling language on a high, independent abstraction level, which of course is convenient for specifying security policies. When this schema is applied and instantiated with concrete parameters (in the dialect) we gain a security design language that is tailored to our problem domain needs (see Figure 4).

[1] I was told there is another example using the Chinese Wall as security concept.
[2] so called *protected resources*

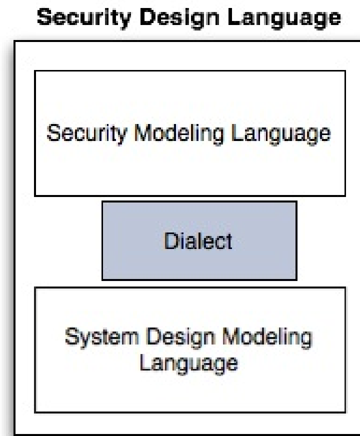


FIGURE 4. Security Design Language Composition Scheme

## 2. MODELING LANGUAGES

A modeling language consists of the following parts:

- abstract syntax:
  defines the language primitives used to build models (model elements)[3]
- concrete syntax (notation):
  defines the graphical representation of the language primitives [4]
- semantics:
  as we deal with formal languages there is a well defined "meaning" of each model element [5]

### 2.1. Security Modeling Language.

The paper defines his own security modeling language that is called "SecureUML" which uses UML as basis. The authors claim that any other language could have been applied but due to the broad acceptance and usage UML seemed to be the best choice for demonstration. SecureUML allows formalizing of access control requirements on protected resources based on the role-based access control principle. It only deals with resources as abstract entities that are referenced inside a specified security policy.

### 2.2. System Design Modeling Language.

This is a domain-specific language for system engineering, which is used to precisely specify system properties. There are also

[3]for SecurityUML they use MOF (Meta-Object Facility), the language that is used for the UML metamodel, to specify the abstract syntax

[4]for SecurityUML a UML Profile (stereotypes, tagged values added to UML) is used to specify the concrete syntax

[5]for SecurityUML a mathematical formalism with first-order structures / formulas together with order-sorted signatures and labeled transition systems is used; semantics serve as basis for model checking, so you could mathematically prove that your model fulfills some properties, but for mds to work semantics is not a requirement

languages which allow the specification of the semantics of a system i.e. the business logic. But that's not what we have here, we only describe the structure of the system. The business logic isn't generated automatically you have to implement functionality by hand. System components, their abilities (interfaces) and the relationships between components may be modeled, but in general no statements about which resources are security sensitive and should be protected can be made. Depending on the perspective this may also be seen as an advantage as a separation of these concerns keeps the system model clean and simple.

2.3. **Dialect.** To identify protected resources and to make statements about them in security policies we have to connect the two worlds of security modeling and system design modeling. Here the dialect comes into play to bridge the two worlds and to help to settle up a consistent security design language. The dialect identifies resources[6] that shall be protected and defines composite actions and their hierarchy to enable high-level security policies.

2.4. **Security Design Language.** As mentioned before a Security Design Language is the result of the combination of a Security Modeling and a System Design Modeling Language with help of a domain specific Dialect. This combination is carried out by a merge of metamodels of both languages. So the abstract syntax metamodel of the system design modeling language is merged with the one of the security modeling. The same happens with the concrete syntax metamodels. We also have to define the semantics of the new generated language. The dialect is the glue in this combination, subtyping is used in the metamodels to integrate the components [7].

## 3. Model Transformation

In order to use the system with the specified properties in an abstract model, a transformation of it into the target platform has to be done. In the examples the target platforms are .NET / EJB / Java Servlet. As mentioned above we don't automatically generate business logic from the model. The generated code implements access control mechanisms corresponding to the security policy we have formalized in our model. This code is marked as read-only and may/should not be edited to ensure consistency. System design issues modeled with the system design modeling language are constructed as empty method stubs, which have to by filled out manually in order to implement the actual business logic of the system. The developer will complete these stubs, omitting the read-only parts [8]. The system with its design and security requirements is developed together in the same process within the same model using the same representation. The paper states the assumption that the design modeling language is already equipped with these transformation functions mapping model primitives to code or system infrastructure. Transformation functions for the security modeling language have to be defined and they have to fulfil the following properties:

[6]in the examples a certain UML stereotype is used

[7]e.g. p.25 in the paper, Resource is the super type for the protected resource Entity, the same happens with Atomic- and CompositeActions

[8]or by using a dedicated IDE the read only code is hidden and may not be changed

- semantic preserving:
  the new security rules don't break the existing rules, so the system behaviour isn't influenced by the new rules. If you think of the system as a final state machine (FSM) no new states or transitions are added but rather a security facility (a security monitor) that checks every state transition before it is executed. If a state transitition breaks the specified security (security policy) the transition is prohibited.
- more detailed:
  the state transitions allowed by the security design semantics, have the same effects on the system state as before the new transformation rules were added.
- correctness of transformation functions:
  a security monitor allows an action if and only if the action is allowed according to the semantics of the security design language.

There are two examples in this paper (ComponentUML,ControllerUML). Note that changing the model triggers a retransformation. Therefore you have to separate your business logic somehow, or the manually inserted code is gone[9]!

## 4. Methodology

The common system developing methodology is slightly changed to carry out the necessary activities to realize the model driven security approach.

- Tool development process:
  A system architect has first of all to specify the security modeling language and the system design modeling language. After a specification of a dialect and the transformation functions a combination of the two languages will yield the security design language.
- System development process:
  The security design language can now be used by software engineers to model a system equipped with corresponding security properties. Then the system infrastructure is automatic generated from the model with help of the transformation functions. The missing business logic is implemented, the system built and tested.

## 5. SecureUML

5.1. **abstract syntax.** As mentioned before the abstract syntax metamodel (see Figure 5)is built with MOF. RBAC serves as basis and is extended with:

- composite containers of subjects
- composite containers of actions

The abstract common super type Subject with its descendants Group and User has an aggregation relation with Group. This composite pattern allows not only the encapsulation of users to groups but also of whole groups (heterogeneous collection of users and groups). The role hierarchy allows roles to grow larger by inheriting the permissions of the ascendant role. The composite pattern is also applied to actions

[9]can be solved with a special IDE or a hook in the method stub that calls the method body that is kept in another file

offering the possibility to specify composite and atomic actions. It is also possible to define hierarchies among composite actions. Permissions are factored out in to the ability to carry out actions on resources. A resource is the base class of all model elements representing protected resources. A permission (representing declarative access control) can be restricted by an authorization constraint that is specified by OCL (Object Constraint Language) expressions representing programmatic access control. The possible actions on resources are represented by the class action. AtomicActions are low-level action, and can be directly mapped to actions of the target platform (e. g "execute" executes a method of the resource) whereas a CompositeAction is rather abstract and doesn't have a direct counterpart. A permission on a CompositeAction always impacts all other contained subordinated actions. Abstract action types are defined in the dialect[10].

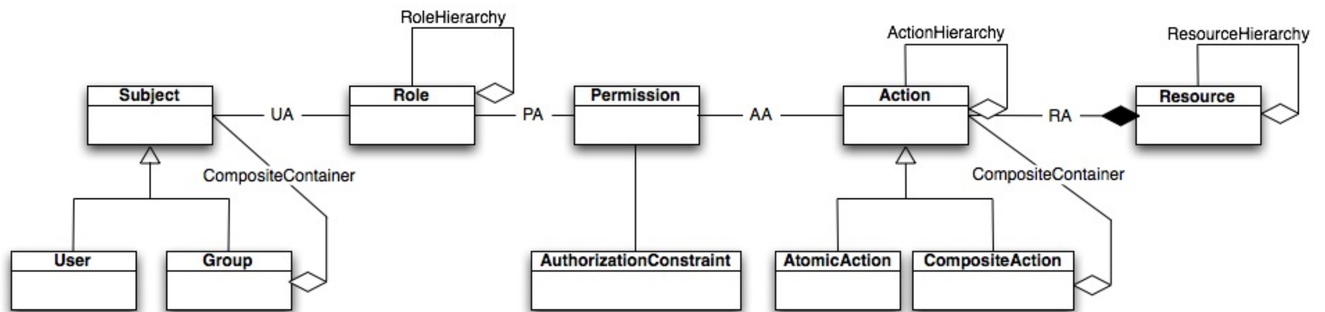


FIGURE 5. SecurityUML, abstract syntax

5.2. **concrete syntax.** SecureUML (as the name implies) uses UML as concrete syntax and is defined through an UML Profile metamodel. A profile extends standard UML with help of stereotypes and tagged values. Stereotypes define new UML elements by extending existing ones with specific properties that are suitable for the particular problem domain (e.g. «interface»). Tagged values are also used for extension. Resources are present after the combination with the security design modeling language. The dialect identifies the protected ones by mapping them to concrete UML stereotypes e.g. «entity». For this UML elements permissions can be formulated with SecureUML. A protected resource has assigned roles. A permission using this resource and this role is attached as association class. Permissions may be formulated on different levels of abstraction. In the example there are permissions formulated on ClassActions (resource scope) and on ClassMethodActions (method scope) e.g. «ClassAction»Meeting:read means that all attributes and all side-effect free methods may be read/executed, «ClassMethodAction»Meeting_notify:execute means that the action notify in the UML class Meeting may be executed. Authorization constraints are specified as comments containing OCL that are attached to the Association Classes.

[10]can be seen as a drawback, because the language designer must foresee all needed composite actions in the model

5.3. **access decisions.** Two kinds of information used to decide on access.

- declaritive access control: static assignments of users and permission to roles (RBAC configuration)
- programmatic access control: run-time data that is used in authorization constraints

With permissions you define the declarative access control. A role may carry out certain actions on a specific protected resource. Therefore permissions cover the static security aspects of a system. In order to consider run time aspects too we need something more. Here AuthorizationConstraints come into play. You can define constraints using OCL (Object Constraint Language) and attach them as UML comments to a permission. The authorization constraints are mapped for example to if-statements inside the EJB methods [11] . If the constraint isn't fulfilled a security exception may be thrown.

## 6. An Example: ComponentUML

Concrete system design modeling language called ComponentUML combined with security modeling language SecureUML. The resulting security design language is used to model the system together with security policies. As mentioned before the combination or merge of both languages together with the dialect is done separately for each language part.

6.1. **extending the abstract syntax.** The abstract syntax of ComponentUML design modeling language is merged with that of SecureUML. The model elements Role and Permission are added to ComponentUML. Now we provide a dialect to identify the protected resources. We select those model elements we would like to control access to them. These protected resources have a generalization association with the SecureUML type Resource. The set of actions that is offered by every element type is defined in ComponentUML. The actions can be grouped in abstract types which then have a generalization relationship with the SecureUML type Composite Action. This allows different levels of abstraction of actions. The actual action hierarchy is defined through OCL constraints for each Composite Action. The introduction of composite actions (and CompisiteAction types) and the hierarchy among them is specified in the dialect.

6.2. **extending the concrete syntax.** analog

6.3. **extending the semantics.** A labeled transition system is used (LTS) over a first-order signature. Mathematical details are omitted here. Interesting properties [12]:

- side-effect free method (read of attribute): state transition $q \rightarrow q', q == q$
- set of attribute: the read of the attribute after the set returns the value used for the set

[11] because there are no method preconditions like in Eiffel

[12] see paper p.28

6.4. **Generating an EJB System, basic generation rules.** Transformation of model elements to EJB constructs:

- entity: entity bean (with empty interfaces / implementations)
- method: declaration goes in the interface of the corresponding entity bean, a method stub goes into the implementation of the interface
- attributes: access through getter setter methods with persistence information for storage in db[13]
- association ends: collection of all entities we have an associations with, as with attributes we use getters and setters
- permissions: (defined for those entities that are protected resources) are mapped to xml-elements that specify the RBAC policy in EJB (more details below)

This is for declarative access control so far. Programmatic access control (authorization constraints on permissions) expressions are transformed to assertions in the corresponding methods at the start of the method body (simple if statement with exception throw block if not fulfilled).
In EJB a permission element names a role and the set of corresponding EJB methods that the members of the role may execute (declarative access control based on RBAC). EJB access control works on method level therefore we have to transform high level policies to these low level permission elements. A high-level composite action *read-all* is mapped to permission elements containing all attribute getter methods and side-effect free methods. Composite actions are expanded to all subordinated actions. Doing this manually without a model transformation is quite error-prone. A misplaced entry could cause a security breach. Another problem is the size and the kind of this task. A programmer "forced" to write this code by hand might take a shortcut by using wildcards (which are allowed in EJB) to reduce the amount of written code. This arguments stress the advantages of an automatic approach. To map the role hierarchy (which isn't supported in EJB) we have to flatten it by generating the reflexive, transitive closure. The same applies to action hierarchies.
**personal comments:**
Because the model is transformed automatically there is no ambiguity in the implementation of the security mechanisms. The security policy may be specified using role hierarchies, action hierarchies and composite containers for users and actions, this abilities of the model allows to keep the model simple, readable and uncluttered. This abilities are not supported by the target platform and therefore mapped to the available primitive abilities through automatic transformation[14]. Doing this by hand using copy / paste or wildcards is really error prone.

6.5. **Correctness of generation.** The informal semantics of the EJB Security architecture could be formulated as follows: An execution of a method is prohibited if either the executing user doesn't have the appropriate role according to the permission (static, declarative access control constraint not fulfilled) or an assertion error occurs during method execution (dynamic, programmatic access control constraint not

[13] because the EJB RBAC security principle is only applied to methods, attributes are declared as private and getters and setters are added

[14] of course you need a code generator based on the transformation functions

fulfilled). The informal semantics of our Security Design Language would be: Access to a protected resource is allowed if and only if the user is assigned to a role that is larger than or equal to a role that has a permission p and this permission refers to an action that is larger than or equal to the atomic action corresponding to executing this method.[15]

[15] There is also a short sketch of an correctness proof see paper p.35