Model Driven Security for Process-Oriented Systems*

David Basin
ETH Zurich
8006 Zurich, Switzerland
basin@inf.ethz.ch

Jürgen Doser Torsten Lodderstedt
University of Freiburg
79110 Freiburg, Germany
{doser,tlodderstedt}@informatik.uni-freiburg.de

ABSTRACT

Model Driven Architecture is an approach to increasing the quality of complex software systems based on creating high-level system models and automatically generating system architectures from the models. We show how this paradigm can be specialized to what we call Model Driven Security. In our specialization, a designer builds a system model along with security requirements, and automatically generates from this a complete, configured security infrastructure.

We propose a modular approach to constructing modeling languages supporting this process, which combines languages for modeling system design with languages for modeling security. We present an application to constructing systems from process models, where we combine a UML-based process design language with a security modeling language for formalizing access control requirements. From models in the combined language, we automatically generate security architectures for distributed applications.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering, Object-oriented design methods*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages, Methodologies, Tools*

General Terms

Design, Languages, Security

Keywords

Model Driven Architecture, Metamodeling, RBAC, Security Engineering, UML

1. INTRODUCTION

Many processes are security critical in the sense that security requirements are a central part of process requirements and security mechanisms are required for their realization. Examples range from the authorization of a military engagement, to an enterprise purchase process, to even the coordination of the sequence of user interface masks displayed to a user.

In such examples, the security policy can be quite complex and may be comprised of a collection of requirements, which are associated with different points of execution and checked and enforced at these points. If we view a process abstractly as being characterized by system *modes* or *states*, then the policy is defined and enforced over a sequence of states in a run of the process. This is because the policy itself may stipulate how the process evolves over time, i.e., what state transitions are allowed, since the transitions permitted depend on information available in the different states (e.g., in an enterprise purchase process), or because the policy itself must be implemented in various steps (e.g., a nuclear launch requiring authorization along a chain of command).

*This work has been supported by the German "Bundesministerium für Wirtschaft und Arbeit" under the reference number IT-MM-01MS107. The authors are responsible for the content of this publication.

SACMAT'03, June 1–4, 2003, Como, Italy.

The above considerations motivate the two questions that we address here. (1) How can we model designs along with their security policies in a process-oriented setting? And (2) how can we make the transition from designs and policies to secure systems?

A partial answer to both questions is given by the approach of *Model Driven Architecture* (MDA) [5], which supports system development by employing a model-centric and generative development process. MDA is based on creating design models and using tool support to automatically generate substantial parts of systems from these models. In the context of process modeling, MDA can be realized by building an automata-oriented (e.g., state chart) model of a system, and using a CASE-tool to automatically generate a process controller (i.e., a state-transition system) from the model.

Our contribution in this paper is to extend the partial answer above to a complete answer by specializing MDA to *model driven security*. Namely, we present an approach to building secure systems where we (1) integrate security models with UML process models and (2) automatically generate executable systems with fully configured security infrastructures from these integrated models. This closes the gap between process models and security models and thereby the gap between software engineering and security engineering. This means that security can be tightly integrated into a system during design, rather than after-the-fact, increasing the security and maintainability of the resulting system.

With respect to (1), we present the security modeling language SecureUML and show how to integrate it with UML process models. SecureUML is a UML-based language for modeling access control requirements that generalizes role-based access control (RBAC) as defined in [4]. SecureUML provides a language for formalizing security policies about protected resources, but leaves open the nature of these resources, i.e., whether they are components, processes, tasks in a process, etc. We show how to combine this generic security policy language with a process modeling language by defining a *dialect*, which identifies process elements (e.g., states, actions, or even processes themselves) as protected resources. As a result, the right to start a process or to execute parts of the process depends on the modeled access control policy. The result is a modeling language capable of formalizing a new kind of model, *security design models*, which integrate security and design requirements.

We have chosen RBAC and UML as foundations for this work because they are well-established, popular standards and well-supported by platforms and tools. By combining these two formalisms, our work enables developers to formalize access control in the context of different kinds of system models using intuitive, graphical UML notation.

With respect to (2), we show how to translate security design models into implementations of controller objects for multi-tier applications augmented by an access control infrastructure enforcing the modeled access control policy. We use the Java Servlet architecture as the target platform of the generation and we have evaluated our approach using a prototype generator that is implemented within the MDA-tool ArcStyler [6].

Related Work. Kandala and Sandhu showed that RBAC can serve as access control model for workflow systems [10] and Atluri and Huang proposed a Workflow Authorization Model [1]. While [10] and our work focus on processes and tasks as protected resources, [1] provides a model for dynamically assigning privileges to subjects during the execution of tasks. In contrast to [10], our modeling language also supports authorization constraints on permissions.

Bertino, Ferrari and Atluri in [2] and Jaeger in [8] emphasized the need for authorization constraints on role and user assignments. In contrast, authorization constraints as defined in SecureUML make the applicability of a permission depend on the system state, e.g. the attribute values of an object or properties of the caller.

There is related work on using UML for security modeling. Epstein and Sandhu showed in [3] how UML can be used to model RBAC policies. Although there are some similarities between our work and theirs (e.g., the focus is on the static aspects of a RBAC model), there are also differences (e.g., they use a different UML notation and document application constraints as preconditions in plain English, whereas we use OCL). Also, we show how UML state machine elements can be incorporated in the definition of an access control policy.

Jürjens proposed in [9] a UML-extension called UMLSec for specifying secure systems. UMLsec allows one to annotate different UML diagrams with security requirements, which can then be validated using formal verification techniques. In contrast, we focus on specifying access control requirements in class and state chart diagrams and develop tool support to automatically generate systems obeying these requirements.

Finally, in [11] we introduced the modeling language SecureUML. In our current work we have both simplified and generalized the formalism and introduced the concept of dialects. The major difference though is that [11] uses class diagrams as a system modeling language and generates security infrastructures for distributed components conforming to the Enterprise JavaBeans standard [12], whereas in the current work we focus on process modeling and generate process controllers. The two works are complementary and together they suggest the wide scope of SecureUML and the idea of model driven security.

Organization. In Section 2, we give a brief overview of relevant aspects of UML and we introduce a UML-based process design language. We define SecureUML in Section 3 and integrate it into our process design language by defining a SecureUML dialect in Section 4. We give an example policy in Section 5 and explain the generation of secure web applications (based on Java Servlet) from modeled policies in Section 6. In Section 7, we report on experience and draw conclusions.

2. THE UNIFIED MODELING LANGUAGE

In UML [13], the structural aspects of systems are defined in terms of classes, each formalizing a set of objects with common services, properties, and behavior. Services are described by methods, properties by attributes and associations, and the behavior can e.g., be characterized by a state machine that is attached to a particular class. Additional constraints can be given using expressions in the Object Constraint Language (OCL).

UML can serve as foundation for building domain specific languages by defining *stereotypes*, which introduce new language primitives by subtyping UML core types, and *tagged values*, which represent new properties of these primitives. Model elements are assigned to such types by labeling them

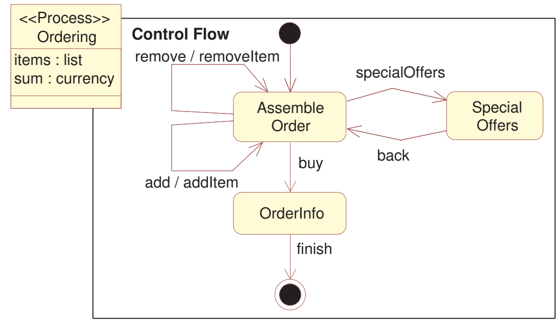

Figure 1: Structure and control flow of the ordering process

with the corresponding stereotype. Below, we give an example of a domain specific language.

UML can also be used as metamodeling language, where UML diagrams are used to formalize the abstract syntax of another (modeling) language. We define SecureUML in this way because it allows us to use object-oriented concepts in our language definition.

Example: A Process Design Language

We now give an example of a process design language using (stereotyped) classes to define process types where, for each process type, a state machine describes the corresponding behavior. Figure 1 shows a model of an ordering process for an online book club. The process type `Ordering` is shown in the upper-left corner of the figure as a UML class with the stereotype «Process». `Ordering` has attributes to store the ordered items and the total cost.

The control flow of the process is defined by the state machine shown in the box labeled "Control Flow". The rectangles and circles represent states and the arrows represent transitions. Transitions may be labeled with the name of the triggering event and (separated by a slash) the name of the action that is executed during state transition.

In our example, `AssembleOrder` is the state where club members can browse through the club catalog, add items to their order, or remove them. The state `SpecialOffers` accesses special offers, which are restricted to "gold members" or to users whose current order is more than 100 €. In the state `OrderInfo`, additional data is processed, like the delivery address.

As the paper proceeds, we will use this as a running example to show how the requirement for restricted access

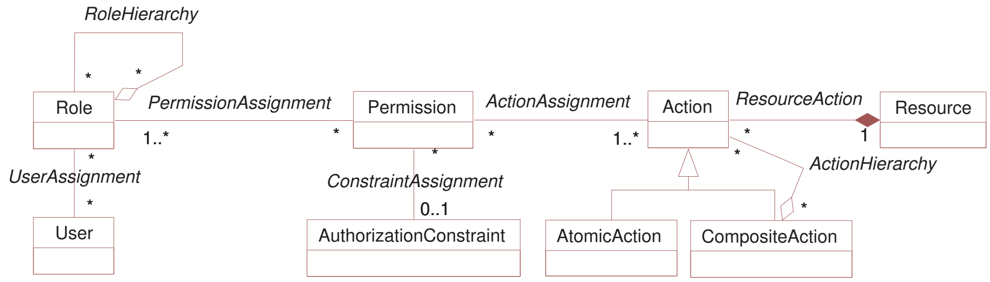

Figure 2: SecureUML Metamodel

to `SpecialOffers` and other requirements are stated as a formal access control policy and how access control infrastructures can be generated that enforce such a policy. SecureUML, which we introduce next, will provide the foundation for expressing such a policy.

3. SecureUML

We now define the syntax and semantics of SecureUML. Moreover, as we will use SecureUML to create visual models (like in UML), we will also endow it with a *notation* (e.g., icons, strings, or geometric figures). To distinguish the two kinds of syntax, we will call the underlying syntax the *abstract syntax* and the notation the *concrete syntax*. We give examples of concrete syntax in Section 5.

3.1 Abstract Syntax

Figure 2 presents the metamodel that defines the abstract syntax of SecureUML. The types User, Role, and Permission and the relations UserAssignment, PermissionAssignment, and RoleHierarchy are directly adopted from the proposed RBAC standard [4]. In the following we focus on our additions.

An AuthorizationConstraint is a logical predicate that is attached to a permission by the association ConstraintAssignment and makes the permission's validity a function of the system state, e.g., dependent on the current time, date, or attribute values. Consider the informal policy given in Section 2, stating that ordinary book club members may only access special offers if their current order is more than 100 €. Such a policy could be formalized by giving a permission for a role `Member` to the state `SpecialOrders`, restricted by an authorization constraint on the attribute `sum` of the ordering process. Such a constraint is given by an OCL expression, where the system model determines the vocabulary that can be used (classes and methods) extended by the additional symbol *caller*, which represents the name of the user on whose behalf an action is performed.

The types Resource and Action roughly correspond to the terms Operation and Object in [4] and formalize a generic resource model that serves as a foundation for integrating SecureUML into system modeling languages.[1] Resource is the base class of all model elements in the system modeling language that represent protected resources, e.g., *processes* in our process design language. Each resource offers one or more actions and each action belongs to exactly one resource, which is denoted by the composite aggregation ResourceAction. Therefore, we can directly assign actions to a permission by the relation ActionAssignment, rather than associating permissions with resource/action pairs.

We differentiate between two categories of actions formalized by the action subtypes AtomicAction and CompositeAction. Atomic actions are low-level actions that can be mapped directly to actions of the target platform, e.g. the action *execute* of a method. In contrast, composite actions are high-level actions that may not have direct counterparts on the target platform. Composite actions, ordered in an ActionHierarchy, are used to group actions and provide a rich modeling vocabulary.

As we will see, the semantics of a permission defined on a composite action is that the right to perform the action implies the right to perform any of the (transitively) contained subactions. This semantics yields a simple basis for defining useful high-level actions. Suppose the security pol-

[1] We use different names to prevent confusion with similar terms used in the object-oriented community.

icy grants a role the permission to "read" an entity. Using an action hierarchy, we can unambiguously formalize this by stipulating that such a permission includes the permission to read the value of every entity attribute and to execute every side-effect-free method of the entity. Another reason for introducing action hierarchies is that they simplify the development of generation rules as it is sufficient to define these rules only for atomic actions.

The concrete resource types, their actions, and the action hierarchy are defined as part of a SecureUML dialect, which integrates SecureUML with a system development language. In Section 4 we will present a dialect that combines SecureUML with our process design language.

3.2 Semantics

SecureUML formalizes access control decisions that depend both on the static assignments of users and permissions to roles and on the satisfaction of authorization constraints in the current system state. The tricky bit in defining the semantics is formalizing the satisfaction of a constraint relative to the system state, which varies over time. To accomplish this, we represent the system state at a time point t by a first-order structure $\mathfrak{S}(t)$. An authorization constraint can be expressed as a formula in first-order logic over a signature Σ, which is determined by the system model. The question of whether a user u is allowed to perform an action a can then be cast as the logical decision problem $\mathfrak{S}(t) \models \phi_{\mathrm{AC}}(u, a)$, formalizing that access should be granted if and only if $\phi_{\mathrm{AC}}(u, a)$ is satisfied in $\mathfrak{S}(t)$. The formula $\phi_{\mathrm{AC}}(u, a)$ is built from the constraints that are assigned to permissions and depends on the static assignments of users and permissions to roles, designated as the *access control configuration*.

The basic elements of the access control configuration are the sets *Users*, *Roles*, *Permissions*, and *Actions*, each containing entries for every model element of the corresponding metamodel types User, Role, Permission, and Action. Over these sets, we have the relations $UA \subseteq \mathit{Users} \times \mathit{Roles}$, $PA \subseteq \mathit{Roles} \times \mathit{Permissions}$, and $AA \subseteq \mathit{Permissions} \times \mathit{Actions}$, which contain tuples for each instance of the corresponding associations[2] in the abstract syntax of SecureUML.

[2]We abbreviate the names of the associations in Figure 2 as their capital letters.

Additionally, we define the partial order $\geq_{Roles}$ as well as the partial order $\geq_{Actions}$, on the sets of roles and actions respectively, where we write superior roles (or actions) on the left (bigger) side of the $\geq$-symbol. $\geq_{Role}$ is given by the reflexive, transitive closure of the aggregation association RoleHierarchy on Role in Figure 2 and $\geq_{Actions}$ is defined analogously based on the aggregation association ActionHierarchy.

Given a system model (i.e., a UML model in our process modeling language), we define a signature $\Sigma = (S, F, P)$, where S is a set of sorts, F is a set of typed function symbols (including constants), and P is a set of typed predicate symbols. S contains one sort for each class in the system model and additionally the sort *Users*, which represents the users of a system as defined in the access control configuration. F contains a function symbol for each attribute or side-effect free method of the model that does not return a Boolean value, and the constant symbol self_{Class} for each $Class$ in the system model denoting the object that is currently accessed. Additionally, there is a constant symbol *caller* denoting the user on whose behalf an action is performed at a time point t. Finally, P contains a predicate symbol for every side-effect free method in the system model that returns a Boolean value.

Given an access control configuration, we define $\phi_{\mathrm{AC}}(u, a)$ by first defining the auxiliary function

$$\mathrm{UAP} : (\mathit{Users} \times \mathit{Actions}) \rightarrow 2^{\mathit{Permissions}}$$

by

$$\begin{aligned} \mathrm{UAP}(u,a) \quad := \quad & \{p \in \mathit{Permissions} \mid \exists r \in \mathit{Roles} : \mathrm{UA}(u,r) \wedge \\ & \exists r' \in \mathit{Roles} : r \geq_{Roles} r' \wedge \mathrm{PA}(p,r') \wedge \\ & \exists a' \in \mathit{Actions} : \mathrm{AA}(p,a') \wedge a' \geq_{Actions} a\} \ , \end{aligned} \tag{1}$$

which determines the permissions a user has for an action. We also introduce the function constraint : $\mathit{Permissions} \rightarrow \mathcal{L}^{\Sigma}$, which maps permissions to their associated constraints translated[3] into the first-order language over Σ, associating the formulae *true* with unconstrained permissions. In this setting, we define that at time t the user u is allowed to

[3]Due to space restrictions we omit the translation rules from OCL to first-order logic.

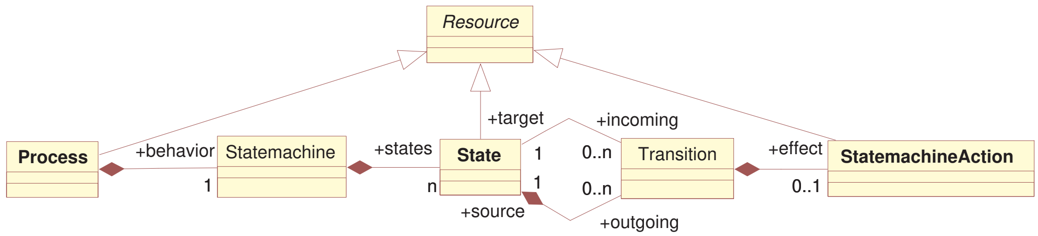

Figure 3: SecureUML Dialect for Process Design

resource type	action	subactions
Process	activate recursive	*activate* for the same process, *activate recursive* for all process states
	activate	-
State	activate recursive	*activate* for the same state, *execute* for all actions on outgoing transitions
	activate	-
StatemachineAction	execute	-

Table 1: Actions and Action Hierarchies of the Dialect

perform the action a if and only if $\mathfrak{S}(t) \models \phi_{\mathrm{AC}}(u,a)$, where

$$\phi_{\mathrm{AC}}(u,a) := \bigvee_{p \in \mathrm{UAP}(u,a)} \mathrm{constraint}(p) \quad . \tag{2}$$

This means that access is granted if and only if the user u has a permission p for which the corresponding formula constraint(p) is valid at time t. Note that the disjunction over an empty set UAP results in the formula *false*.

4. A DIALECT FOR PROCESS DESIGN

As previously noted, SecureUML is general in that it leaves open the nature of protected resources. The general scheme for combining SecureUML with a system design language is to formalize a *dialect*, which identifies primitives of the system modeling language as SecureUML resources and assigns them atomic and composite actions (which are given by an action hierarchy). Hence, to combine SecureUML with the proposed process design language, we define a dialect where we interpret a process type and the elements of its state machine as a hierarchy of protected resources and additionally define appropriate resource actions and an action hierarchy.

The abstract syntax of this dialect is defined by the metamodel shown in Figure 3. We fix the elements Process, State and StatemachineAction as the protected resources, which is denoted by defining types for these primitives as subtypes of the base type Resource of the SecureUML metamodel (see Section 3.1).

We stipulate the actions offered by the resource types and the structure of the action hierarchy, shown in Table 1, by defining OCL invariants on the resource types. For example, we define that each Process has the actions *activate* and *activate recursive* and that the composite action *activate recursive* for a process recursively includes the actions listed on all states of the process.

Action hierarchies, here and in general, support the definition of natural and concise security policies. For example, an *activate recursive* permission on a process grants a role the privilege to perform all actions on the process. Alternatively, the right to execute a single state machine action can be defined with the action *execute*.

5. AN EXAMPLE POLICY

In this section we explain our UML-based notation and illustrate the semantics of SecureUML using our running example.

5.1 An Example UML Model

In Figure 4, we formalize a security policy where (1) all members of the club can place orders, (2) members having

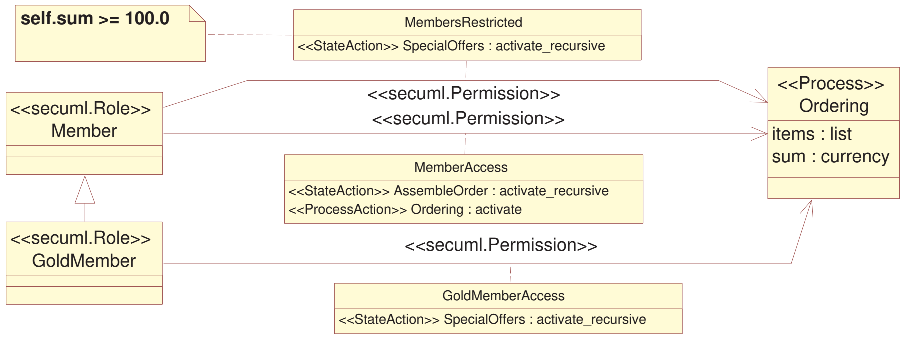

Figure 4: Access control policy for the ordering process

the "gold" status are given special offers, and (3) ordinary members may also access special offers, but only after they have ordered items for more than 100 €.[4]

We start by declaring the roles `Member` and `GoldMember`, which are represented by classes with the stereotype «secuml.Role» and define `GoldMember` as a subrole of `Member`. Next, we define several permissions, each formalizing a requirement of the informal policy description. As Figure 4 suggests, a permission is represented by an association class with the stereotype «secuml.Permission» connecting a role with a UML class representing a protected resource, designated as the *root resource* of this permission. Each attribute of the association class represents the assignment of an action to the permission, where the action is identified by a resource name and an action name. The action name is given by the attribute type, e.g. "activate". The resource name is stored in the tagged value identifier and references the root resource or one of its subresources. Its format depends on the referenced resource's type, which is determined by the stereotype of the attribute. For example, an attribute with the stereotype «StateAction» and the identifier "AssembleOrder" denotes an action on the corresponding state of the process "Ordering". The stereotypes for action references are defined as part of the dialect; our process design language has the stereotypes "ProcessAction", "StateAction", and "ActionAction", which are respectively used to label references to actions on processes, states, and actions.

The first requirement of our example policy is formalized by the permission `MemberAccess` that grants the role `Member` the right to perform the actions *activate* on `Ordering` and *activate recursive* on the state `AssembleOrder`. The permission `GoldMemberAccess` grants the right to activate the state `SpecialOffers` to the role `Member` and in doing so formalizes the second requirement of our policy. The third requirement is formalized by the permission `MemberRestricted`, which is augmented with an authorization constraint. The permission itself grants the role `Member` the right to perform *activate recursive* on the state `SpecialOffers`. The constraint

```
self.sum >= 100.0
```

restricts this permission to cases where an order is more than 100 €. This expression is defined in a tagged value of the permission class, but we show it here in the figure as a text box attached to this class.

5.2 Example Semantics

We now illustrate our semantics by analyzing several access control decisions for the users *Alice* and *Bob*, each trying to perform the action `activate` on the state `SpecialOffers`. The (partial) example access control configuration is

$$\begin{aligned}
\text{Users} &:= \{\text{Bob}, \text{Alice}\} \\
\text{Roles} &:= \{\texttt{Member}, \texttt{GoldMember}\} \\
\text{Permissions} &:= \{\texttt{MemberRestricted}, \texttt{GoldMemberAccess}, \ldots\} \\
\text{Actions}^{5} &:= \{\texttt{SpecialOffers.activate}, \\
&\quad\ \texttt{SpecialOffers.activateRecursive}, \ldots\} \\
\text{UA} &:= \{(\text{Bob}, \texttt{Member}), (\text{Alice}, \texttt{GoldMember})\}
\end{aligned}$$

[4]Note that for sake of simplicity we omit additional arrangements, which would handle the case that the total sum for ordinary items falls below 100 € after adding special items.

[5]Actions are denoted by the name of their resource and their name, separated by a dot.

$$
\begin{aligned}
\mathrm{PA} &:= \{(\texttt{Member}, \texttt{MemberRestricted}),\\
&\quad (\texttt{GoldMember}, \texttt{GoldMemberAccess}), \ldots\}\\
\mathrm{AA} &:= \{(\texttt{MemberRestricted},\\
&\quad \texttt{SpecialOffers.activateRecursive}),\\
&\quad (\texttt{GoldMemberAccess},\\
&\quad \texttt{SpecialOffers.activateRecursive}), \ldots\}\\
\geq_{Roles} &:= \{(\texttt{GoldMember}, \texttt{Member})\}\\
\geq_{Actions} &:= \{(\texttt{SpecialOffers.activateRecursive},\\
&\quad \texttt{SpecialOffers.activate}), \ldots\}\ ,
\end{aligned}
$$

and the signature Σ, derived from the system model, is

$$
\begin{aligned}
S &:= \{\mathit{Processes}, \mathit{Real}\} \cup \{\texttt{Users}\}\\
F &:= \{\mathit{self}_{\mathit{Processes}}, \mathit{sum}\} \cup \{\texttt{caller}\}\\
P &:= \{\geq_{Real}\}\ .
\end{aligned}
$$

The constant symbol $\mathit{self}_{\mathit{Processes}}$ of sort *Processes* denotes the currently accessed process and $\mathit{sum} : \mathit{Processes} \to \mathit{Real}$ represents the value of the attribute `sum` of a process instance. The predicate symbol $\geq_{Real}$ represents the standard order over the real numbers.

Suppose that Alice wants to perform the action explained above at time t_1. The corresponding structure $\mathfrak{S}(t_1)$ over the signature Σ is given by the interpretation

$$
\begin{aligned}
\texttt{caller}^{\mathfrak{S}(t_1)} &:= \text{Alice}\\
\mathit{Processes}^{\mathfrak{S}(t_1)} &:= \{\mathit{process}_{\text{Alice}}, \mathit{process}_{\text{Bob}}\}\\
\mathit{self}_{\mathit{Processes}}^{\mathfrak{S}(t_1)} &:= \mathit{process}_{\text{Alice}}\\
\mathit{sum}^{\mathfrak{S}(t_1)} &:= \{(\mathit{process}_{\text{Alice}}, 30), (\mathit{process}_{\text{Bob}}, 55)\}\ .
\end{aligned}
$$

The formula that has to be satisfied by the structure $\mathfrak{S}(t_1)$ in order to grant Alice access is built according to the definition (2), given in Section 3.2. The constraint

```
self.sum >= 100
```

on the permission `MemberRestricted` is translated into the formula

$$\mathit{sum}(\mathit{self}_{\mathit{Processes}}()) \geq_{Real} 100$$

in the language defined by the signature Σ, and the formula for the permission `GoldMember` is *true*. The access decision is formalized as

$$\mathfrak{S}(t_1) \models \mathit{true} \vee (\mathit{sum}(\mathit{self}_{\mathit{Processes}}()) \geq_{Real} 100)\ ,$$

which is satisfied.

Alternatively, suppose that Bob tries to perform this action at time t_2. The corresponding structure $\mathfrak{S}(t_2)$ differs from $\mathfrak{S}(t_1)$ by the interpretation of the constant symbols $\mathit{self}_{\mathit{Processes}}$ and `caller`, now referring to $\mathit{process}_{\text{Bob}}$ and Bob, respectively. However, Bob only has the permission `MemberRestricted` for this action; hence

$$\mathfrak{S}(t_2) \models \mathit{sum}(\mathit{self}_{\mathit{Processes}}()) \geq_{Real} 100$$

is required for access. Since Bob only has items that sum to 55 € in his order, this constraint is not satisfied and access should be denied here.

6. GENERATION

We have implemented a prototype generator for the construction of secure web applications within the MDA-tool ArcStyler [6]. This tool already provides a transformation function for converting UML classes and state machines into controller classes for web-applications, executed in a Java Servlet environment. Hence our task was to extend this function to generate a security infrastructure from SecureUML model elements.

Java Servlet [7] is a popular technology for developing web applications as it provides a rich environment for programming dynamic web pages. This technology supports RBAC; however its URL-based authorization scheme only enforces access control when a request arrives from outside the web server. This is ill-suited for modern web applications that are built from multiple servlets, with one acting as the central entry point to the application. This entry point servlet acts as a dispatcher in that it receives all requests and forwards them (depending on the application state) to the other servlets, which execute the business logic. The standard authorization mechanism only provides protection for the dispatcher. To overcome this weakness, we generate access control infrastructures that exploit the programmatic access control mechanism which servlets provide, where the role assignments of a user can be retrieved by any servlet.

We augmented ArcStyler's transformation function by a generation rule that produces Java assertions and adds them as preconditions to the methods for *process activation*, *state activation*, and *action execution*. These assertions are of the form

$$\texttt{if (!(}\quad \bigvee_{p \in \mathrm{AP}(a)} \Big(\Big(\bigvee_{r \in PR(p)} \mathrm{userrole}(r) \Big) \wedge \mathrm{constraint}(p) \Big) \texttt{))}$$

$$\texttt{c.forward("/unauthorized.jsp");}$$

This rule is similar to Equation (2), which defines $\phi_{\mathrm{AC}}(u, a)$ in Section 3.2, as each permission represents an (alternative) authorization to execute an action. However, here we must consider all permissions for all users on an action, which are determined by the function AP. For each permission, there is first a check whether a user is assigned to one of the roles in PR, which denotes the set of all roles assigned to the permission. Because servlets do not support role inheritance, the transitive closure of all roles associated with a permission must be determined using the association RoleHierarchy. If a constraint is assigned to the permission, it is evaluated afterwards. The request is forwarded to an error page by the term `c.forward("/unauthorized.jsp")` if access is denied. As an example, the assertion generated for `SpecialOffers.activate` is:

```
if (!(
    /** Permission GoldMemberAccess **/
    request.isUserInRole("GoldMember")
     ||
    /** Permission MemberRestricted **/
    ((request.isUserInRole("Member") ||
      request.isUserInRole("GoldMember"))
      && this.getSum() >= 100.0)
    )) c.forward("/unauthorized.jsp");
```

The role check is performed using the method *isUserInRole(Role)* on the request object and each constraint is translated into a Java expressions, which accesses the attributes and side-effect free methods of the controller. For example, the OCL constraint

```
self.sum >= 100.0
```

is translated into the Java expression

```
this.getSum() >= 100.0
```

as shown above.

Comparing the code generated above with the example given in Section 5.2 shows that the declarative semantics of our models is preserved (under the operational semantics of Java) by the generation function. The user *Alice* is granted unrestricted access to the state by the first part of the assertion, whereas the right for the user *Bob* depends on the return value of the method `getSum()`, which is used to obtain the current value of the controller's attribute `sum`.

7. ANALYSIS AND CONCLUSIONS

We have carried out a number of small and medium scale experiments, specifying controllers for applications like online shopping and banking. Our experiments show that the combination of UML process models with a vocabulary based on RBAC, authorization constraints, and action hierarchies is expressive enough to naturally and concisely describe complex security policies. Hence, our proposal provides a concrete example of how to close the gap between design models (here process models) and security models.

The generative approach taken also closes another gap: the one between system design and implementation. Doing so has a number of advantages. First, it guarantees the conformance of the implementation with the model. Second, it eases system maintenance (as the process and security policy are clearly and formally documented) and evolution (as changes are easily made at the model level and automatically propagated to the implementation). And finally, it enhances portability since models are technology independent and hence the migration to new technologies can be realized by changing the generation rules, not the models themselves. It should be a simple matter to develop other transformation functions for translating models into secure, executable systems based on other technologies like workflow management systems or other web application platforms.

There are a several promising areas for future work. A natural step is to increase the expressiveness of either SecureUML or the design modeling language with which it is integrated. An example of the former would be adding primitives for modeling other security aspects, like digital signatures or auditing (and then, of course, generating corresponding infrastructures). An example of the latter would be supporting a more comprehensive process design language. The idea of joining languages via a dialect, as sketched in Section 4, provides a way of tackling these problems independently, and then combining solutions. Another promising direction is to leverage the fact that our security design models have a well-defined semantics. Hence, it should be possible to carry out automatic property checking of security design models to detect and correct design errors and even to verify the correctness of the model transformation process itself.

8. REFERENCES

[1] V. Atluri and W.-K. Huang. An authorization model for workflows. In *Proceedings of the Fifth European Symposium on Research in Computer Security, Rome, Italy*, volume 1146 of *LNCS*, pages 44–64. Springer, 1996.

[2] E. Bertino, E. Ferrari, and V. Atluri. An approach for the specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information Systems Security*, 2(1):65–104, February 1999.

[3] P. Epstein and R. S. Sandhu. Towards a UML Based Approach to Role Engineering. In *Proceedings of 4th ACM Workshop on Role-Based Access Control*, pages 145–152, 1999.

[4] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[5] D. S. Frankel. *Model Driven Architecture^{TM}: Applying MDA^{TM} to Enterprise Computing.* John Wiley & Sons, 2003.

[6] R. Hubert. *Convergent Architecture: Building Model Driven J2EE Systems with UML*. John Wiley & Sons, Inc., 2001.

[7] J. Hunter. *Java Servlet Programming, 2nd Edition.* O'Reilly & Associates, 2001.

[8] T. Jaeger. On the increasing importance of constraints. In *Proceedings of 4th ACM Workshop on Role-Based Access Control*, pages 33–42, 1999.

[9] J. Jürjens. UMLsec: extending UML for secure systems development. In *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

[10] S. Kandala and R. Sandhu, editors. *Secure Role-Based Workflow Models*, volume 215 of *IFIP Conference Proceedings.* Kluwer, 2002.

[11] T. Lodderstedt, D. A. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.

[12] R. Monson-Haefel. *Enterprise JavaBeans (3rd Edition).* O'Reilly & Associates, 2001.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.