MODEL DRIVEN SECURITY

David Basin,[1] Jürgen Doser,[1] and Torsten Lodderstedt[2]

[1]*ETH Zürich, Switzerland**

[2]*Interactive Objects Software GmbH, Germany*

Abstract We present a new approach to building secure systems. In our approach, which we call Model Driven Security, designers specify system models along with their security requirements and use tools to automatically generate system architectures from the models including complete, configured security infrastructures. Rather than fixing one particular modeling language for this process, we propose a general schema for constructing such languages that combines languages for modeling systems with languages for modeling security. We present several instances of this schema that combine (both syntactically and semantically) different UML modeling languages with a security modeling language for formalizing access control requirements. From models in the combined languages, we automatically generate security architectures for distributed applications, built from declarative and programmatic access control mechanisms. We have implemented this approach and report on a case-study with the resulting tool.

1. Introduction

Model building is standard practice in software engineering. The construction of models during requirements analysis and system design can improve the quality of the resulting systems by providing a foundation for early analysis and fault detection. The models also serve as specifications for the later development phases and, when the models are sufficiently formal, they can provide a basis for refinement down to code.

Model building is also carried out in security modeling and policy specification. However, its integration into the overall development process is problematic and suffers from two gaps. First, security models

*This work has been partially supported by the Swiss "Federal Office for Education and Science" in the context of the EU-funded Integrated Project TrustCoM (IST-2002-2.3.1.9 Contract-No. 1945). The authors are responsible for the content of this publication.

and system design models are typically disjoint and expressed in different ways (e.g., security models as structured text versus graphical design models in languages like UML). In general, the integration of system design models with security models is poorly understood and inadequately supported by modern software development processes and tools. Second, although security requirements and threats are often considered during the early development phases (requirements analysis), and security mechanisms are later employed in the final development phases (system integration and test), there is a gap in the middle. As a result, security is typically integrated into systems in a post-hoc manner, which degrades the security and maintainability of the resulting systems.

In this paper, we take up the challenge of providing languages, methods, and tools for bridging these gaps. Our starting point is the concept of *Model Driven Architecture* (MDA) [Frankel, 2003], which has been proposed as model-centric and generative approach to software development. Conceptually, the MDA approach has three parts: (1) developers create *system models* in high-level modeling languages like UML; (2) tools are used to perform *automatic model transformation*; and the result is (3) a *target (system) architecture*. Whereas the generation of simple kinds of code skeletons by CASE-tools is now standard (e.g., generating class hierarchies from class diagrams), Model Driven Architecture is more ambitious and aims at generating nontrivial kinds of system infrastructure from models.

Our main contribution is to show how the Model Driven Architecture approach can be specialized to what we call *Model Driven Security* by extending its three parts: system design models are extended with security requirements and model transformation is extended to generate security infrastructure for the target system. The most difficult part of this specialization is the first, concerning the models themselves, and here we propose a general schema for combining languages for security modeling with those for design modeling. Our schema provides a recipe for language combination at the level of both syntax and semantics, for example providing sufficient conditions for the combination to be semantically well-defined. The main idea is to define security modeling languages that are general in that they leave open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, states in a controller, etc. Such a security modeling language can then be combined with a system design modeling language by defining a *dialect*, which identifies elements of the design language as the protected resources of the security language. In this way, we can define families of languages that flexibly combine design modeling languages

and security modeling languages, and are capable of formulating system designs along with their security requirements.

To show the feasibility of this approach and to illustrate some of the design issues, we present several detailed examples. First, we specify a security modeling language for modeling access control requirements that generalizes Role-Based Access Control (RBAC) [Ferraiolo et al., 2001]. To support visual modeling, we embed this language within an extension of UML and hence we call the result *SecureUML*. Afterwards, we give two examples of design modeling languages, one based on class diagrams and the other based on statecharts. We then combine each of these with SecureUML by defining dialects that identify particular elements of each design modeling language as protected SecureUML resources.

In each case, we define model transformations for the combined modeling language by augmenting model transformations for the UML-based modeling languages with the additional functionality necessary for translating our security modeling constructs. The first dialect provides a language for modeling access control in a distributed object setting and we define a transformation function that produces security infrastructures for distributed systems conforming to the Enterprise JavaBeans (EJB) standard. The second dialect provides a language for modeling security requirements for controllers for multi-tier architectures and the transformation function generates access control infrastructures for web applications.

As a proof of concept, within the MDA-tool ArcStyler [Hubert, 2001] we have built a prototypical generator that implements the above mentioned transformation functions for both dialects. We report on this, as well as on experience with our approach. Overall, we view the result as a large step towards integrating security engineering into a model-driven software development process. This bridges the gap between security analysis and the integration of security mechanisms into end systems. Moreover, it integrates security models with system design models and thus yields a new kind of model, *security design models*.

2. Background

We first introduce a design problem along with its security requirements that will serve as a running example throughout this paper. Afterwards, we introduce the modeling and technological foundations that we build upon: the Unified Modeling Language, Model Driven Architecture, Role-based Access Control, and several security architectures.

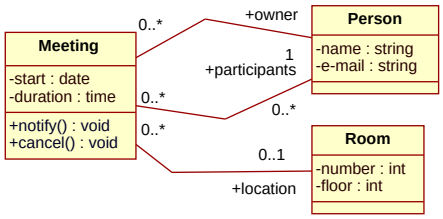

Figure 1. Scheduler Application Class Diagram

2.1 A Design Problem

As a running example, we will consider developing a simplified version of a system for administrating meetings. The system should maintain a list of users (we will ignore issues such as user administration) and records of meetings. A meeting has an owner, a list of participants, a time, and a place. Users may carry out standard operations on meetings such as creating, reading, editing, and deleting them. A user may also cancel a meeting, which deletes the meeting and notifies all participants by email.

As the paper proceeds, we will see how to formalize a design model for this system along with the following security policy.

1 All users can create new meetings and read all meeting entries.

2 Only the owner of a meeting may change meeting data and cancel or delete the meeting.

3 A supervisor can cancel any meeting.

2.2 The Unified Modeling Language

The Unified Modeling Language (UML) [Rumbaugh et al., 1998] is a widely used graphical language for modeling object-oriented systems. The language specification differentiates between *abstract syntax* and *notation* (also called *concrete syntax*). The abstract syntax defines the language primitives used to build models, whereas the notation defines the graphical representation of these primitives as icons, strings, or figures. UML supports the description of the structure and behavior of systems using different model element types and corresponding diagram types. In this paper, we focus on the model element types comprising class and statechart diagrams.

The structural aspects of systems are defined using classes, e.g., as in Figure 1, which models the structure of our scheduling application.

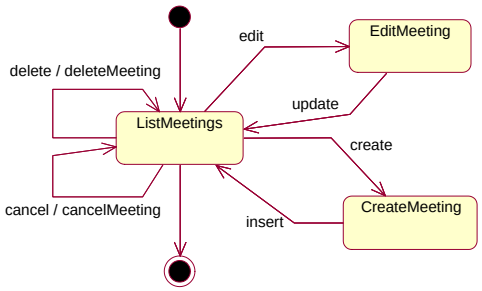

Figure 2. Scheduler Application Statechart

This model consists of three classes: Meeting, Person, and Room. A Meeting has attributes for storing the start date and the planned duration. The participants and the location of the meeting are specified using the association ends participants and location. The method notify notifies the participants of changes to the schedule. The method cancel cancels the meeting, which includes notifying the participants and canceling the room reservation.

In contrast, state machines describe the behavior of a system or a class in terms of states and events that cause a transition between states. Figure 2 shows the statechart diagram for our scheduling application. In the state ListMeetings, a user can browse the scheduled meetings and can initiate (e.g., by clicking a button in a graphical user interface) the editing, creation, deletion, and cancellation of meetings. An event of type edit causes a transition to the state EditMeeting, where the currently selected meeting (stored in ListMeetings) is edited. An event of type create causes a transition to the state CreateMeeting, where a new meeting is created from data entered by the user. An event of type delete in state ListMeetings triggers a transition that executes the action deleteMeeting, where the currently selected meeting is deleted from the database. Similarly, an event of type cancel causes the execution of cancelMeeting, which calls the method cancel on the selected meeting.

UML also provides a specification language called OCL, the *Object Constraint Language*. OCL expressions are used to formalize invariants for classes, preconditions and postconditions for methods, and guards for enabling transitions in a state machine. As an example, we can add to the class Meeting in Figure 1 the following OCL constraint, stating that the participants of a meeting must always include the meeting's owner.

```
context Meeting inv:
  self.participants->includes(self.owner)
```

2.3 Model Driven Architecture

Model Driven Architecture (MDA) has been proposed as an approach to specifying and developing applications where systems are represented as models and transformation functions are used to map between models as well as to automatically generate executable code [Frankel, 2003]. Of course, the fully automatic synthesis of complex systems from high-level descriptions is unobtainable in its full generality. We cannot, in general, automatically generate the functions implementing a specification of a system's functional behavior, i.e., its "business logic". But what is possible is to automate the generation of platform-specific support for different kinds of non-functional system concerns, such as support for persistence, logging, and the like, i.e., system *aspects*, in the aspect-oriented programming sense [Kiczales et al., 1997], that cut across different system components. Our work shows that security, in particular access control, is one such aspect that can be automatically generated and that this brings with it many advantages.

The starting point of MDA is the use of domain-specific languages to formalize models for different application domains or system aspects. In our work, we define modeling languages by directly formalizing their metamodels. As a metalanguage, we use the Meta-Object Facility (MOF), which is essentially a subset of UML that is well-suited for formalizing metamodels using standard object-oriented concepts like class and inheritance. MOF provides a more expressive formalism for defining modeling languages than other alternatives, e.g., the use of UML profiles or conventional definition techniques like the Backus-Naur Form (BNF). For example, in MOF, we can directly formalize relations between model primitives, which is one of the key ideas we use when combining modeling languages (see, for example, the discussion on subtyping in Section 5.1). MOF also offers advantages for building MDA tools. There is tool support for automatically creating repositories and maintaining metadata based on MOF, e.g. [Akehurst and Kent, 2002]. Moreover, by separating the abstract syntax of languages from their UML-based concrete syntax (defined by UML profiles), we can concisely define modeling languages and directly use UML CASE-tools for building models.

2.4 RBAC

Mathematically, access control expresses a relation AC between a set of *Users* and a set of *Permissions*:

$$AC \subseteq \mathit{Users} \times \mathit{Permissions}\,.$$

User u is granted permission p if and only if $(u, p) \in AC$. Aside from the technical question of how to integrate this relation into systems so that

granting permissions respects this relation, a major challenge concerns how to effectively represent this information since directly storing all the (u, p) pairs scales poorly. Moreover, this view is rather "flat" and does not support natural abstractions like sets of permissions.

Role-Based Access Control, or RBAC, addresses both of the above limitations. The core idea of RBAC is to introduce a set of roles and to decompose the relation AC into two relations: user assignment UA and permission assignment PA, i.e.,

$$UA \subseteq Users \times Roles, \qquad PA \subseteq Roles \times Permissions \ .$$

The access control relation is then simply the composition of these relations:

$$AC = PA \circ UA \, .$$

To further reduce the size of these relations and support additional abstraction, RBAC also has a notion of hierarchy on roles. Mathematically, this is a partial order $\geq$ on the set of roles, with the meaning that larger roles inherit permissions from all smaller roles. Formally, this means that the access control relation is now given by the equation

$$AC = PA \circ \geq \circ \, UA \ \ ,$$

where the role hierarchy relation $\geq$ is also part of the composition. To express the same access control relation without a role hierarchy, one must, for example, assign each user additional roles, i.e., a user is then not just assigned his original roles, but also all smaller roles. The introduction of a hierarchy, like the decomposition of relations, leads to a more expressive formalism in the sense that one can express access control relations more concisely. Role hierarchies also simplify the administration of access control since they provide a convenient and intuitive abstraction that can correspond to the actual organizational structure of companies.

We have chosen RBAC as a foundation of our security modeling language because it is well-established and it is supported by many existing technology platforms, which simplifies the subsequent definition of the transformation functions. However, RBAC also has limitations. For example, it is difficult to formalize access control policies that depend on dynamic aspects of the system, like the date or the values of system or method parameters. We have extended RBAC with authorization constraints to overcome this limitation. Furthermore, although many technologies support RBAC, they differ in details, like the degree of support for role-hierarchies and the types of protected resources. As we will see later, our approach of generating architectures from models provides a means to overcome such limitations and differences in technologies.

2.5 Security Architectures

We make use of two different security architectures in this paper. We provide an overview of them here, focusing on their support for access control.

Enterprise JavaBeans Enterprise JavaBeans (EJBs) is a component architecture standard [Monson-Haefel, 2001] for developing server-side components in Java. These components usually form the business logic of multi-tier applications and run on application servers. The standard specifies infrastructures for system-level aspects such as transactions, persistence, and security. To use these, an EJB developer declares properties for these aspects, which are managed by the application server. This configuration information is stored in *deployment descriptors*, which are XML documents that are installed together with the components.

The access control model of EJB is based on RBAC, where the protected resources are the methods accessed using the interfaces of an EJB. This provides a mechanism for *declarative access control* where the access control policy is configured in the deployment descriptors of an EJB component. The security subsystem of the EJB application server is then responsible for enforcing this policy on behalf of the components. The following example shows a permission definition that authorizes the role Supervisor to execute the method cancel on the component Meeting.

```
<method-permission>
  <role-name>Supervisor</role-name>
  <method>
    <ejb-name>Meeting</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>cancel</method-name>
    <method-params/>
  </method>
</method-permission>
```

As this example illustrates, permissions are defined at the level of individual methods. A `method-permission` element lists one or more roles using elements of type `role-name` and one or more EJB methods using elements of type `method`. An EJB method is identified by the name of its component (`ejb-name`), the interface it implements (`method-intf`), and the method signature (`method-name` and `method-params`). The listed roles are granted the right to execute the listed methods.

EJB offers the additional possibility of enforcing access control within the business logic of components. This mechanism is called *programmatic access control* and is based on inserting Java assertions in the methods

of the bean class. To support this, EJB provides interfaces for retrieving security relevant data of a caller, like his name or roles.

Java Servlets The Java Servlet Specification [Hunter, 2001] specifies an execution environment for web components, called *servlets*. A servlet is basically a Java class running in a web server that processes http requests and creates http responses. Servlets can be used to dynamically create HTML pages or to control the processing of requests in large web applications.

The execution environment, called the *servlet container*, supports both declarative and programmatic access control. For declarative access control, permissions are defined at the level of uniform resource locators (URLs) in XML deployment descriptors. Programmatic access control is used to determine the identity and the roles of a caller and to implement decisions within a servlet.

3. Model Driven Security: an Overview

At the heart of Model Driven Security are security design models, which combine security and design requirements. Rather than presenting one particular modeling language for constructing these models, we propose a schema for building such languages in a modular way. The overall form of our schema is depicted in Figure 3. The schema is parameterized by three languages:

1 a *security modeling language* for expressing security policies;

2 a *system design modeling language* for constructing design models; and

3 a *dialect*, which provides a bridge by defining the connection points for integrating (1) with (2), e.g., model elements of (2) are classified as protected resources of (1).

This schema defines a family of security design languages. By different instantiations of the three parameters, we can build different languages, tailored for expressing different kinds of designs and security policies.

To automate our approach to Model Driven Security, for each schema instance, we define transformation functions that map models to security infrastructures. This must be done on a case-by-case basis, but, like with compilers, the implementation is just a one-time cost and the result is a general tool.

Below we discuss these aspects in more detail. However, due to space limitations, we will focus on one particular security modeling language, which we call *SecureUML*, that is based on an extension of Role-Based

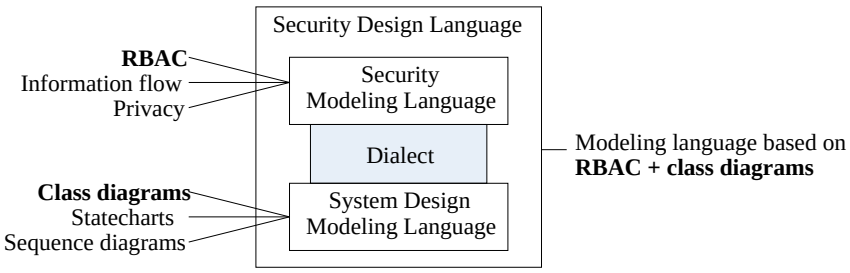

Figure 3. Security Design Language Schema

Access Control. We will present this language in detail, emphasizing the general metamodeling ideas behind it. We will later present two different system design modeling languages and different dialects.

3.1 Security Modeling Languages

A security modeling language is a formal language in that it has a well-defined *syntax* and *semantics*. As we intend these languages to be used for creating intuitive, readable models (e.g., visual models, like in UML), they will also be employed with a *notation*. To distinguish these two kinds of syntax, and following UML (cf. Section 2.2), we call the underlying syntax the *abstract syntax* and the notation the *concrete syntax*. In general, the abstract syntax is defined formally, e.g., by a grammar, whereas the notation is defined informally. The translation between notation and abstract syntax is generally straightforward; we give examples in Section 4.2.

Designing modeling languages is a creative and nontrivial task, in particular when it comes to their semantics and developing (semantics-preserving) transformation functions. However, it is not our expectation that each application developer must also be a language designer. This task will be done once and for all for a large class of applications by security and system architects. We will use SecureUML to illustrate that it is possible to design security modeling languages that are general, usable with different design modeling languages, and applicable to a wide scope of problems.

The definition of a language's abstract syntax will be based on MOF and the concrete syntax will be defined by a UML profile. In Section 4 we explain this in detail as well as the semantics of SecureUML and language combination. Note that the abstract syntax and semantics of SecureUML define a modeling language for access control policies that is independent of UML and which could be combined with design

modeling languages different from those of UML. However, we do make a commitment to UML when defining notation, and our use of a UML profile to define a UML notation motivates the name SecureUML.

3.2 System Design Languages and Dialects

In our approach, a system design modeling language is merged with a security modeling language by merging their vocabularies at the levels of notation and abstract syntax. But more is required: it must be possible to build expressions in the combined language that combine subexpressions from the different languages. That is, security policies expressed in the security modeling language must be able to make statements about system resources or attributes specified in the design modeling language. It is the role of the dialect to make this connection. We will show one way of doing this using subtyping (in the object-oriented sense) to classify constructs in one language as belonging to subtypes in the other. We will provide examples of such combinations in Section 5 and Section 7.

These ideas are best understood on an example. Our security modeling language SecureUML provides a language for specifying access control policies for actions on protected resources. However, it leaves open what the protected resources are and which actions they offer to clients. These depend on the primitives for constructing models in the system design modeling language. For example, in a component-oriented modeling language, the resources might be methods that can be executed. Alternatively, in a process-oriented language, the resources might be processes with actions reflecting the ability to activate, deactivate, terminate, or resume the processes. Or, if we are modeling file systems, the protected resources might correspond to files that can be read, written, or executed. The dialect specifies how the modeling primitives of SecureUML are integrated with the primitives of the design modeling language in a way that allows the direct annotation of model elements representing protected resources with access control information. Hence it provides the missing vocabulary to formulate security policies involving these resources by defining:

- the model element types of the system design modeling language that represent protected resources;
- the actions these resource types offer and hierarchies classifying these actions; and
- the default access control policy for actions where no explicit permission is defined (i.e., whether access is allowed or denied).

We give examples of integrating SecureUML into different system modeling languages in Sections 5.1 and 7.1.

3.3 Model Transformation

Given a language that is an instance of the schema in Figure 3, we must define a transformation function operating on models constructed in the language. As our focus in this paper is on security, we shall assume that the system design modeling language used is already equipped with a transformation function, consisting of transformation rules that define how model elements are transformed into code or system infrastructure. Our task then is to define how the additional modeling constructs, from the security modeling language, are translated into system constructs. Our aim here is neither to develop nor to generate new kinds of security architectures, but rather to capitalize on the existing security mechanisms of the target component architecture and generate appropriate instances of these mechanisms. Of course, for this to be successful, the modeling constructs in the security modeling language and their semantics should be designed with an eye open to the class of architectures and security mechanisms that will later be part of the target platforms. This requires care during the language design phase.

We will illustrate the transformation process using SecureUML and its combination with two different design languages. In one case, we define a transformation function that translates component models into secure systems based on the component platform EJB (Section 6). In the other case, our transformation function maps controller models into secure web applications based on the Java Servlet standard (Section 7).

4. SecureUML

We now define the abstract syntax, concrete syntax, and semantics of SecureUML. While we will later give examples of how to combine SecureUML syntactically with different design modeling languages, we describe here the semantic foundations for this combination.

4.1 Abstract Syntax

Figure 4 presents the metamodel that defines the abstract syntax of SecureUML. The left-hand part of the diagram basically formalizes an extension of RBAC, where we extend *Users* (defined in Section 2.4) by *Groups* and formalize the assignment of users and groups to roles by using their common supertype Subject. The right-hand part of the diagram factors permissions into the ability to carry out *actions* on *resources*. Permissions may be constrained to hold only in certain system states

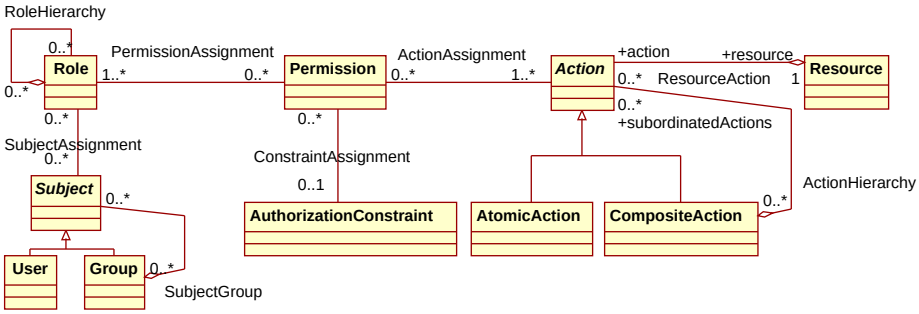

Figure 4. SecureUML Metamodel

by *authorization constraints*. Additionally, we introduce hierarchies not only on roles (which is standard for RBAC), but also on actions.

Let us now examine these types and associations in more detail. Subject is the base type of all users and groups in a system. It is an abstract type (type names in *italic font* in class diagrams represent abstract types), which means that it cannot be instantiated directly: each subject is either a user or a group. A User represents a system entity, like a person or a process, whereas a Group names a set of users and groups. Subjects are assigned to groups by the aggregation SubjectGroup, which represents an ordering relation over subjects. Subjects are assigned to roles by the association SubjectAssignment.

A Role represents a job and bundles all privileges needed to carry out the job. A Permission grants roles access to one or more actions, where the actions are assigned by the association ActionAssignment and the entitled roles are denoted by the association PermissionAssignment. Due to the cardinality constraints on these associations, a permission must be assigned to at least one role and action. Roles can be ordered hierarchically, which is denoted by the aggregation RoleHierarchy, with the intuition that the role at the part end of the association inherits all the privileges of the aggregate.

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. Consider a policy stating that an employee is allowed to withdraw money from a company account provided the amount is less than $5,000. Such a policy could be formalized by giving a permission to a role Employee for the method withdraw, restricted by an authorization constraint on the parameter amount of this method. Such constraints are given by OCL expressions, where

the system model determines the vocabulary (classes and methods) that can be used, extended by the additional symbol caller, which represents the name of the user on whose behalf an action is performed.

Resource is the base class of all model elements in the system modeling language that represent protected resources. The possible operations on these resources are represented by the class Action. Each resource offers one or more actions and each action belongs to exactly one resource, which is denoted by the composite aggregation ResourceAction. We differentiate between two categories of actions formalized by the action subtypes AtomicAction and CompositeAction. Atomic actions are low-level actions that can be directly mapped 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 are ordered in an ActionHierarchy.

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 one of the (transitively) contained subordinated actions. This semantics yields a simple basis for defining high-level actions. Suppose that a security policy grants a role the permission to "read" an entity. Using an action hierarchy, we can formalize this by stating 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. Action hierarchies also simplify the development of generation rules since it is sufficient to define these rules only for the atomic actions.

Together, the types Resource and Action formalize a generic resource model that serves as a foundation for combining SecureUML with different system modeling languages. The concrete resource types, their actions, the action hierarchy, and the rules for deriving resources along an inheritance hierarchy are defined as part of a SecureUML dialect.

4.2 Concrete Syntax

SecureUML's concrete syntax is based on UML. To achieve this, we define a UML profile that formalizes the modeling notation of SecureUML using stereotypes and tagged values. In this section, we introduce the modeling notation and explain how models in concrete syntax are transformed into abstract syntax.

Table 1 gives an overview of the mapping between elements of the SecureUML metamodel and UML types. Note that a permission, its associations to other elements, and its optional authorization constraint are represented by a single UML association class. Also note that the profile does not define an encoding for all SecureUML elements. For

UML metamodel type and stereotype		SecureUML metamodel type
Class	«User»	User
Class	«Group»	Group
Dependency	«SubjectGroup»	SubjectGroup
Dependency	«SubjectAssignment»	SubjectAssignment
Class	«Role»	Role
Generalization between classes with stereotype	«Role»	RoleHierarchy
AssociationClass	«Permission»	Permission, PermissionAssignment, ActionAssignment, AuthorizationConstraint, and ConstraintAssignment

Table 1. Mapping Between SecureUML Concrete and Abstract Syntax

example, the notation for defining *resources* is left open and must be defined by the dialect. Also, no representation for *subjects* is given because Subject is an abstract type.

We now illustrate the concrete syntax and the mapping to abstract syntax with the example given in Figure 5, which formalizes the second part of the security policy introduced in Section 2.1: only the owner of a meeting may change meeting data and cancel or delete the meeting.

In the SecureUML profile, a role is represented by a UML class with the stereotype «Role» and an inheritance relationship between two roles is defined using a UML generalization relationship. The role referenced by the arrowhead of the generalization relationship is considered to be the superrole of the role referenced by the tail, and the subrole inherits all access rights of the superrole. In our example, we define the two roles User and Supervisor. Moreover, we define Supervisor as a subrole of User.

Users are defined as UML classes with the stereotype «User». The assignment of a subject to a role is defined as a dependency with the stereotype «SubjectAssignment», where the role is associated with the arrowhead of the dependency. In our example, we define the users Alice and Bob, and formalize that Alice is assigned to the role Supervisor, whereas Bob has the role User.[1]

The right-hand part of Figure 5 specifies a permission on a protected resource. Specifying this is only possible after having combined SecureUML with an appropriate design modeling language. The concrete syntax of SecureUML is generic in that every UML model element type can represent a protected resource. Examples are classes, attributes, and methods, as well as state machines and states. A SecureUML dialect specializes the base syntax by stipulating which elements of the system design language represent protected resource and defines the mapping

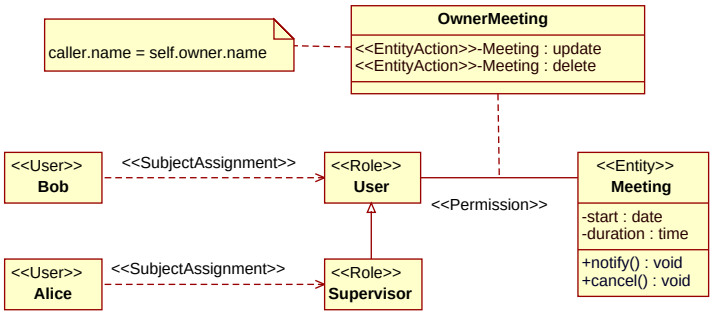

Figure 5. Example of the Concrete Syntax of SecureUML

between the UML representation of these elements and the resource types in the abstract syntax of the dialect. For this example, we employ a dialect (explained in Section 5.1) that formalizes that UML classes with the stereotype «Entity» are protected resources possessing the actions *update* and *delete*, i.e., the class Meeting is a protected resource.

A permission, along with its relations to roles (PermissionAssignment) and actions (ActionAssignment), is defined in a single UML model element, namely an association class with the stereotype «Permission». The association class connects a role with a UML class representing a protected resource, which is designated as the *root resource* of the permission. The actions such a permission refers to may be actions on the root resource or on subresources of the root resource. In our example, the class Meeting is the root resource of the permission OwnerMeeting granted to the role User.

Each attribute of the association class represents the assignment of an action to the permission (ActionAssignment), where the action is identified by the name of its resource and the action name. The action name is given as the attribute's type, e.g. "update". The resource name is stored in the tagged value identifier and references the root resource or one of its subresources. The format of the identifier depends on the type of the referenced resource and is determined by the stereotype of the attribute.

The stereotypes for action references and the naming conventions for identifiers are defined as part of the dialect. As a general rule, the resource identifier is always specified relative to the root resource. This prevents redundant information in the model and inconsistencies when the root resource's name is changed. For example, the attribute start would be referenced by the string "start" and the root resource itself would be referenced by an empty string. Note that the name of the

action reference attribute has only an illustrative meaning. We generally use names that provide information about the referenced resource. In our example, the attribute of type "update" with the stereotype «EntityAction» and the name "Meeting" denotes the action *update* on the class Meeting. As we will later see in Table 2, the permission to *update* an Entity also comprises the permission to *execute* any non-side-effect free method of the Entity, for example the method `cancel()` of the class Meeting. The second attribute in our example denotes the action *delete* on the class Meeting. Together, these two attributes specify the permission to update (which includes canceling) and delete a meeting.

Each authorization constraint is stored as an OCL expression in the tagged value constraint of the permission that it constrains. To improve the readability of a model, we attach a text note with the constraint expression to the permission's association class. In our example, the permission UserMeeting is constrained by the authorization constraint `caller.name = self.owner.name`, which restricts the permission to update and delete a meeting to the owner of the meeting.

4.3 Semantics

The General Idea SecureUML formalizes access control decisions that depend on two kinds of information.

1 Declarative access control decisions that depend on static information, namely the assignments of users and permissions to roles, which we designate as a *RBAC configuration*.

2 Programmatic access control decisions that depend on dynamic information, namely the satisfaction of authorization constraints in the current system state.

While formalizing the semantics of RBAC configurations is straightforward, formalizing the satisfaction of authorization constraints in system states is not. This is mainly because what constitutes a system state is defined by the design modeling language, and not by SecureUML. Since the semantics of SecureUML depends on the set of states, we parameterize the SecureUML semantics by this set. Also, we have to define the semantics of RBAC configurations in a way that supports its combination with the semantics of authorization constraints.

The basic ideas are as follows. To formalize 1, declarative access control decisions, we represent a RBAC configuration as a first-order structure $\mathfrak{S}_{RBAC}$, and we define the semantics of declarative access control decisions by $\mathfrak{S}_{RBAC} \models \phi_{RBAC}(u, a)$, where $\phi_{RBAC}(u, a)$ formalizes that the user u is "in the right role" to perform the action a.

To formalize 2, we represent system states st by (corresponding) first-order structures $\mathfrak{S}_{st}$, and authorization constraints as first-order formulas $\phi^p_{ST}(u)$ (independent of the state st). In accordance with the SecureUML metamodel, constraints are associated with permissions (not actions), and this formula formalizes under which condition the user u has the permission p. Whether this condition holds or not in the state st is then cast as the logical decision problem $\mathfrak{S}_{st} \models \phi^p_{ST}(u)$.

To combine both RBAC configurations and authorization constraints, we combine the first-order structures $\mathfrak{S}_{st}$ and $\mathfrak{S}_{RBAC}$, as well as the first-order formulas $\phi^p_{ST}(u)$ and $\phi_{RBAC}(u, a)$, and use this to formalize the semantics of individual access control decisions. Since the addition of access control changes the run-time behavior of a system, we must also define how the semantics of SecureUML changes the behavior specified by the design modeling language. To accomplish this, we require that the system behavior can be defined by a transition system and we interpret the addition of access control as restricting the system behavior by removing transitions from this transition system. In what follows, we formalize these ideas more precisely.

Declarative Access Control First, we define an order-sorted signature $\Sigma_{RBAC} = (\mathcal{S}_{RBAC}, \leq_{RBAC}, \mathcal{F}_{RBAC}, \mathcal{P}_{RBAC})$ that defines the type of structures that specify role-based access control configurations.[2] Here $\mathcal{S}_{RBAC}$ is a set of sorts, $\leq_{RBAC}$ is a partial order on $\mathcal{S}_{RBAC}$, $\mathcal{F}_{RBAC}$ is a sorted set of function symbols, and $\mathcal{P}_{RBAC}$ is a sorted set of predicate symbols. In detail, we define

$$\mathcal{S}_{RBAC} = \{\mathit{Users}, \mathit{Subjects}, \mathit{Roles}, \mathit{Permissions}, \mathit{AtomicActions}, \mathit{Actions}\} \ ,$$

where $\mathit{Users} \leq_{RBAC} \mathit{Subjects}$, and $\mathit{AtomicActions} \leq_{RBAC} \mathit{Actions}$,

$$\mathcal{F}_{RBAC} = \emptyset \ ,$$

$$\mathcal{P}_{RBAC} = \left\{ \begin{array}{ll} \geq_{\mathit{Subjects}}\!:\mathit{Subjects} \times \mathit{Subjects}, & \mathit{UA}\!:\mathit{Subjects} \times \mathit{Roles}, \\ \geq_{\mathit{Roles}} \ :\mathit{Roles} \times \mathit{Roles}, & \mathit{PA}\!:\mathit{Roles} \times \mathit{Permissions}, \\ \geq_{\mathit{Actions}}\!:\mathit{Actions} \times \mathit{Actions}, & \mathit{AA}\!:\mathit{Permissions} \times \mathit{Actions} \end{array} \right\} \ .$$

The subsort relation $\leq_{RBAC}$ is used here to formalize that *Users* is a subsort of *Subjects* and *AtomicActions* is a subsort of *Actions*.

The predicate symbols UA, PA, and AA denote assignment relations, corresponding in the SecureUML metamodel to the associations SubjectAssignment, PermissionAssignment, and ActionAssignment respectively. The predicate symbols $\geq_{\mathit{Subjects}}$, $\geq_{\mathit{Roles}}$, and $\geq_{\mathit{Actions}}$ denote hierarchies on the respective sets and correspond to the aggregation associations SubjectGroup, RoleHierarchy, and ActionHierarchy respectively.

A SecureUML model defines a Σ_{RBAC}-structure $\mathfrak{S}_{RBAC}$ in the obvious way: the sets *Subjects*, *Users*, *Roles*, *Permissions*, *Actions*, and *AtomicActions* each contain entries for every model element of the corresponding metamodel types Subject, User, Role, Permission, Action, and AtomicAction. Also, the relations UA, PA, and AA contain tuples for each instance of the corresponding association in the abstract syntax of SecureUML.

Additionally, we define the partial orders $\geq_{Subjects}$, $\geq_{Roles}$, and $\geq_{Actions}$ on the sets of subjects, roles, and actions respectively. $\geq_{Subjects}$ is given by the reflexive closure of the aggregation association SubjectGroup in Figure 4 and formalizes that a group is larger than all its contained subjects. $\geq_{Role}$ is defined analogously based on the aggregation association RoleHierarchy on Role and we write subroles (roles with additional privileges) on the left (larger) side of the $\geq$-symbol. $\geq_{Actions}$ is given by the reflexive closure of the composition hierarchy on actions, defined by the aggregation ActionHierarchy. We write $a_1 \geq_{Actions} a_2$, if a_2 is a subordinated action of a_1. These relations are partial orders because aggregations in UML are transitive and antisymmetric by definition.

Note that compared to Figure 4, we have excluded the metamodel types Group, CompositeAction, Resource, and AuthorizationConstraint. Resource is excluded because the target of access control is the actions performed on resources, and not resources themselves. Group and CompositeAction are excluded because groups and composite actions are just subsets of subjects and actions respectively and do not play any further role in the semantics. AuthorizationConstraint is excluded because its semantics is not part of declarative access control, but rather part of programmatic access control.

We define the formula $\phi_{RBAC}(u, a)$ with variables u of sort *Users* and a of sort *Actions* by

$$\begin{aligned}\phi_{RBAC}(u,a) = \exists s \in Subjects, r_1, r_2 \in Roles, p \in Permissions, a' \in Actions.\\ s \geq_{Subjects} u \wedge UA(s, r_1) \wedge r_1 \geq_{Roles} r_2 \wedge\\ PA(r_2, p) \wedge AA(p, a') \wedge a' \geq_{Actions} a \ ,\end{aligned}$$

or equivalently, by factoring out the permissions explicitly, as

$$\phi_{RBAC}(u,a) = \bigvee_{\{p \in Permissions\}} \phi_{User}(u,p) \wedge \phi_{Action}(p,a) \ , \qquad (1)$$

where

$$\begin{aligned}\phi_{User}(u,p) = \exists s \in Subjects, r_1, r_2 \in Roles.\\ s \geq_{Subjects} u \wedge UA(s, r_1) \wedge r_1 \geq_{Roles} r_2 \wedge PA(r_2, p)\end{aligned}$$

states that the user u has the permission p, and

$$\phi_{Action}(p, a) = \exists a' \in Actions.\, AA(p, a') \wedge a' \geq_{Actions} a$$

states that p is a permission for the action a. This is essentially a reformulation of the usual RBAC semantics (cf. Section 2.4). The reason for the factorization given by definition (1) will become clear when we combine this formula with programmatic access control formulas $\phi^p_{ST}(u)$.

The declarative access control part of SecureUML is now defined by saying that a user u may perform an action a only if $\mathfrak{S}_{RBAC} \models \phi_{RBAC}(u, a)$ holds.

Programmatic Access Control While declarative access control decisions can be made independently of the system model, we must explicitly incorporate the syntax and semantics of the design modeling language into SecureUML for programmatic access control. In order to be able to combine the semantics of SecureUML with the semantics of system design modeling languages, we make some assumptions about the nature of the latter, so that the semantic combination will be well-defined.

To make programmatic access control decision, we require that the system design model provides a vocabulary for talking about the structure of the system. More formally, we require that the system design model provides a sorted first-order signature $\Sigma_{ST} = (\mathcal{S}_{ST}, \mathcal{F}_{ST}, \mathcal{P}_{ST})$. Typically, $\mathcal{S}_{ST}$ contains one sort for each class in the system model, $\mathcal{F}_{ST}$ contains a function symbol for each attribute and for each side-effect free method of the model, and $\mathcal{P}_{ST}$ contains predicate symbols for 1-to-many and many-to-many relations between classes. How exactly this signature is defined depends on the semantics of the system design modeling language. We do however require that $\mathcal{S}_{ST}$ contains a sort *Users* and that $\mathcal{F}_{ST}$ contains a constant symbol *caller* of sort *Users*, and a constant symbol $self_C$ for each class C in the system model. This amounts to the requirement that the design modeling language provides some way of talking about *who* is accessing *what*, which is a minimal requirement for any reasonable notion of access control. For practical reasons, we also assume that $\mathcal{F}_{ST}$ contains a function symbol *UserName*, which maps users to a string representation of their names. How the symbols in Σ_{ST} are interpreted in $\mathfrak{S}_{st}$ is again defined by the system design modeling language. Here we only require that the constant symbol $self_C$ is interpreted by the currently accessed object, in case the currently accessed object is of the sort C, and that the constant symbol *caller* is interpreted by the user that initiated this access.

In this setting, the state of the system at a particular time defines a Σ_{ST}-structure $\mathfrak{S}_{st}$. Constraints on the system state $\mathfrak{S}_{st}$ can be ex-

pressed as logical formulas ϕ_{ST}, whereby constraint satisfaction is just the question of whether $\mathfrak{S}_{st} \models \phi_{ST}$ holds.[3]

Combining Declarative and Programmatic Access Control To formalize combined declarative and programmatic access control decisions, we combine the states $\mathfrak{S}_{st}$ and $\mathfrak{S}_{RBAC}$ into the composite structure $\mathfrak{S}_{AC} = \langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{st} \rangle$ and combine the formulas ϕ_{ST} and ϕ_{RBAC} into a new formula ϕ_{AC}. The combined access control decision is then defined as the question of whether $\mathfrak{S}_{AC} \models \phi_{AC}$ holds.

By $\langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{st} \rangle$ we mean that $\mathfrak{S}_{AC}$ is the structure that consists of the carrier sets, functions and predicates from both $\mathfrak{S}_{RBAC}$ and $\mathfrak{S}_{st}$, where we identify the carrier sets of the sort *Users*, which belongs to both structures. As for ϕ_{AC}, observe that authorization constraints are not global constraints, but are attached to permissions (as can be seen in Figure 5) and hence are only relevant for the roles that have these permissions. We denote the authorization constraint that is attached to a permission p by ϕ^p_{ST}, and require that ϕ^p_{ST} is an expression in the first-order language defined by Σ_{ST}. In order to define the language for the combined formula ϕ_{AC}, we combine the signatures Σ_{RBAC} and Σ_{ST} by taking their componentwise union[4], i.e., $\Sigma_{AC} = (\mathcal{S}_{AC}, \leq_{AC}, \mathcal{F}_{AC}, \mathcal{P}_{AC})$, where $\mathcal{S}_{AC} = \mathcal{S}_{RBAC} \cup \mathcal{S}_{ST}$, $\leq_{AC} = \leq_{RBAC} \cup id_{S_{ST}}$, $\mathcal{F}_{AC} = \mathcal{F}_{RBAC} \cup \mathcal{F}_{ST}$, and $\mathcal{P}_{AC} = \mathcal{P}_{RBAC} \cup \mathcal{P}_{ST}$. Here we assume that the signatures Σ_{RBAC} and Σ_{ST} are disjoint, with the exception of the sort *Users*, which belongs to both signatures. Observe that under this definition of Σ_{AC}, $\mathfrak{S}_{AC}$ is a Σ_{AC}-structure.

The combined access control semantics is now defined by the formula

$$\phi_{AC}(u, a) = \bigvee_{p \in Permissions} \phi_{User}(u, p) \wedge \phi_{Action}(p, a) \wedge \phi^p_{ST}(u) \ , \qquad (2)$$

stating that a user u must have a permission p for the action a according to the RBAC configuration and that the corresponding authorization constraint for this permission p must evaluate to *true* for the user u.

Behavioral Semantics of Access Control The preceding paragraphs defined how access control decisions are made in a system state. But what is interesting in the end is how the system behaves when an access control decision is made. In order to define this, we again make some minimal assumptions on the semantics of the design modeling language. Namely, we assume that the semantics of the system design modeling languages can be expressed as a Labeled Transition System (LTS) $\Delta = (Q, A, \delta)$. In this LTS, the set of nodes Q consists of Σ_{ST}-structures, the edges are labeled with elements from a set of actions A that is a superset of

AtomicActions, and $\delta \subseteq Q \times A \times Q$ is the transition relation. Note that we do not require that $A = \mathit{AtomicActions}$ because the design modeling language may define state-changing actions (i.e., those with side-effects) that are not protected. The behavior of the system is defined by the paths (also called traces) in the LTS as is standard: a trace $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ defines a possible behavior if and only if $(s_i, a_i, s_{i+1}) \in \delta$, for $0 \leq i$.

In this setting, adding access control to the system design corresponds to deleting traces from the LTS, i.e., when an action is not permitted then the transition must not be made, and when an action is permitted, the subsequent state must be the same as before adding access control.

More formally, adding access control to a system description means transforming the LTS $\Delta = (Q, A, \delta)$ to an LTS $\Delta_{AC} = (Q_{AC}, A_{AC}, \delta_{AC})$ as follows:

- Q_{AC} is defined by combining system states with RBAC configurations, i.e., $Q_{AC} = Q_{RBAC} \times Q$, where Q_{RBAC} denotes the universe of all finite Σ_{RBAC}-structures.
- A_{AC} is unchanged: $A_{AC} = A$.
- δ_{AC} is defined by restricting δ to the permitted transitions:

$$\begin{aligned}\delta_{AC} = \{(\langle q_{RBAC}, q\rangle, a, \langle q_{RBAC}, q'\rangle) \mid (q, a, q') \in \delta \wedge \\ (a \in \mathit{AtomicActions} \rightarrow \langle q_{RBAC}, q\rangle \models \phi_{AC}(\mathit{caller}, a)\}\end{aligned}$$

  Note that this definition implies that the RBAC configuration does not change during system execution. We do not address issues like run-time user administration in this work.

We will see concrete semantic combinations of SecureUML with different design modeling languages in Sections 5.3 (for ComponentUML) and in Section 7.3 (for ControllerUML).

5. An Example Modeling Language: ComponentUML

In this section, we give an example of a system design language, which we call ComponentUML, and present its combination with SecureUML. We also show how to model security policies using the resulting security design modeling language and we illustrate its semantics using the example introduced in Section 2.1.

ComponentUML is a simple language for modeling distributed object-oriented systems. The metamodel for ComponentUML is shown in Figure 6. Elements of type Entity represent object types of a particular domain. An entity may have multiple methods and attributes, represented

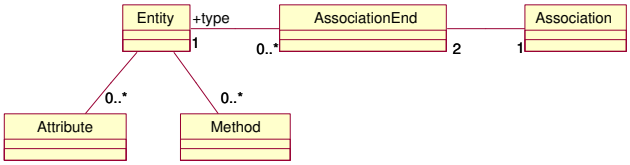

Figure 6. ComponentUML Metamodel

by elements of the types Method and Attribute respectively. Associations are used to specify relations between entities. An association is built from an Association model element and every entity participating in an association is connected to the association by an AssociationEnd.

ComponentUML uses a UML-based notation where entities are represented by UML classes with the stereotype «Entity». Every method, attribute, or association end owned by such a class is automatically considered to be a method, attribute, or association end of the entity, so no further stereotypes are necessary.

Figure 7 shows the structural model of our scheduling application in the ComponentUML notation. Instead of classes, we now have the three entities Meeting, Person, and Room, each represented by a UML class with the stereotype «Entity».

5.1 Extending the Abstract Syntax

Merging the Syntax As the first step towards making ComponentUML security aware, we extend its abstract syntax with the vocabulary of SecureUML by integrating both metamodels, i.e., we merge the abstract syntax of both modeling languages. This is achieved by importing the SecureUML metamodel into the metamodel of ComponentUML. This extends ComponentUML with the SecureUML modeling constructs, e.g., Role and Permission. The use of packages and corresponding namespaces for defining these metamodels ensures that no conflicts arise during merging.

Identifying Protected Resources Second, we identify the model elements of ComponentUML representing protected resources and formalize this as part of a SecureUML dialect. To do this, we must determine which model element we wish to control access to in the resulting systems. When doing this, we must account for what can ultimately be protected by the target platform. Suppose, for example, we decide to interpret entity attributes as protected resources and the target platform supports

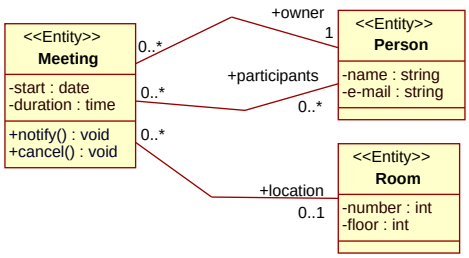

Figure 7. Scheduling Application

access control on methods only. This is possible, but it necessitates a transformation function that transforms each modeled attribute into a private attribute and generates (and enforces access to) access methods for reading and changing the value of the attribute in the generated system.

In our example, we identify the following model elements of ComponentUML as protected resources: Entity, Method, Attribute, and AssociationEnd. This identification is made by using inheritance to specify that these metatypes are subtypes of the SecureUML type Resource, as shown in Figure 8. In this way, the metatypes inherit all properties needed to define authorization policies. Additionally, we define in this figure several action classes as subtypes of the SecureUML class CompositeAction. The action composition hierarchy is then defined as part of each action's type information, by way of OCL invariant constraints (see below) on the respective types.

Defining Resource Actions In the next step, we define the set of actions that is offered by every model element type representing a protected resource, i.e., we fix the domain of the metamodel association ResourceAction for each resource type of the dialect. Actions can be freely defined at every level of abstraction. One may choose just to leverage the actions that are present in the target security architecture, e.g., the action "execute" on methods. Alternatively one may define actions at a higher level of abstraction, e.g., "read" access to a component. This results in a richer, easier to use vocabulary since granting read or write access to an entity is more intuitive than giving someone the privilege to execute the methods `getBalance`, `getOwner`, and `getId`. High-level actions also lead to concise models. We usually define actions of both kinds and connect them using hierarchies.

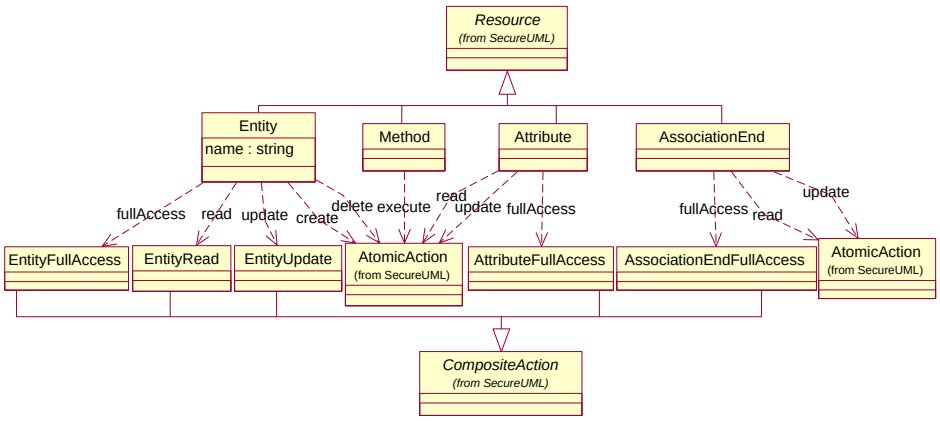

Figure 8. SecureUML Dialect for ComponentUML Metamodel

In the metamodel, the set of actions each resource type offers is defined by the named dependencies from the resource type to action classes, as shown in Figure 8. Each dependency represents one action of the referenced action type in the context of the resource type, where the dependency name determines the name of the action. For example, the metamodel in Figure 8 formalizes that an Attribute always possesses the action *fullAccess* of type AttributeFullAccess and the actions *read* and *update* of type AtomicAction.

Defining the Action Hierarchy As the final step in defining our SecureUML dialect, we define a hierarchy on actions. We do this by restricting the domain of the SecureUML association ActionHierarchy on each composite action type of the dialect by an OCL invariant. An overview of the composite actions of the SecureUML dialect for ComponentUML is given in Table 2. The approach we take is shown for the action class EntityFullAccess by the following OCL expression.

```
context EntityFullAccess inv:
subordinatedActions =
  resource.actions->select(name="create" or name="read" or
                           name="update" or name="delete")
```

This expression states that the composite action EntityFullAccess is larger (a "super-action") in the action hierarchy than the actions *create*, *read*, *update*, and *delete* of the entity the action belongs to.

composite action type	subordinated actions
EntityFullAccess	*create*, *read*, *update*, and *delete* of the entity.
EntityRead	*read* for all attributes and association ends of the entity, and *execute* for all side-effect free methods of the entity.
EntityUpdate	*update* for all attributes of the entity, *update* for all association ends of the entity, and *execute* for all non-side-effect free methods of the entity.
AttributeFullAccess	*read* and *update* of the attribute.
AssociationEndFullAccess	*read* and *update* of the association end.

Table 2. SecureUML Dialect Action Hierarchy

5.2 Extending the Concrete Syntax

In the previous section, we have seen how the abstract syntax of ComponentUML can be augmented with syntax for security modeling by combining it with the abstract syntax of SecureUML. We extend the concrete syntax of ComponentUML analogously by importing the SecureUML notation into ComponentUML. Afterwards, we define well-formedness rules on SecureUML primitives that restrict their use to those ComponentUML elements representing protected resources. For example, the scope of a permission, which is any UML class in the SecureUML notation (see Section 4.2), is restricted to UML classes with the stereotype «Entity». Finally, as shown in Table 3, we define the action reference types for entities, attributes, methods, and association ends.

5.3 Extending the Semantics

Our combination schema requires that we define the semantics of ComponentUML as a labeled transition system $\Delta = (Q, A, \delta)$ over a first-order signature Σ_{ST}. Intuitively, every entity defines a sort in the first-order signature, and every atomic action defined by the SecureUML dialect for ComponentUML (cf. Figure 8) defines an action in the labeled transition system. Side-effect free actions give rise to function and predicate symbols in the first-order signature.

To make this more precise, given a model in the ComponentUML language, we define the signature $\Sigma_{ST} = (\mathcal{S}_{ST}, \mathcal{F}_{ST}, \mathcal{P}_{ST})$ as follows:

- Each Entity e gives rise to a sort S_e in $\mathcal{S}_{ST}$. Additionally, $\mathcal{S}_{ST}$ contains the sorts Users, String, Int, Real, and Boolean:

  $$\mathcal{S}_{ST} = \{S_e \mid e \text{ is an entity}\} \cup \{\mathit{Users}, \texttt{String}, \texttt{Int}, \texttt{Real}, \texttt{Boolean}\} \ .$$

stereotype	resource type	naming convention
EntityAction	Entity	empty string
MethodAction	Method	method signature
AttributeAction	Attribute	attribute name
AssociationEndAction	AssociationEnd	association end name

Table 3. Action Reference Types for ComponentUML

- Each side-effect free entity method m (which is marked in UML by the tagged value "isQuery", set to true) gives rise to a function symbol f_m in $\mathcal{F}_{ST}$ of the corresponding type. Corresponding type here means, in particular, that we add the sort of the entity as an additional parameter, i.e., the "this-pointer" is passed as an additional argument. Each entity attribute at gives rise to a function symbol get_{at} in $\mathcal{F}_{ST}$ (the "get-method") of type $s \to v$, where s is the sort of the entity and v is the sort of the attribute's type. Each association end ae with multiplicity $\{1\}$ gives rise to a function symbol f_{ae}. Finally, we have a constant symbol *caller* of type *Users* and a function symbol *UserName* of type $\mathit{Users} \to \mathit{String}$:

$$\begin{aligned}\mathcal{F}_{ST} = {} & \{f_m \mid m \text{ is an entity method}\} \cup \\ & \{get_{at} \mid at \text{ is an entity attribute}\} \cup \\ & \{\mathit{self}_e \mid e \text{ is an entity}\} \cup \\ & \{f_{ae} \mid ae \text{ is an association end with multiplicity } \{1\}\} \cup \\ & \{\mathit{caller}, \mathit{UserName}\} \ .\end{aligned}$$

- Each association end ae with a multiplicity other than $\{1\}$ gives rise to a binary predicate symbol P_{ae} in $\mathcal{P}_{ST}$ of the type of the involved entities:

$$\mathcal{P}_{ST} = \{P_{ae} \mid ae \text{ is an association end with multiplicity} \neq \{1\}\} \ .$$

We now define the labeled transition system $\Delta = (Q, A, \delta)$ by:

- Q is the universe of all possible system states, which is just the set of all first-order structures over the signature Σ_{ST} that consist of finitely many objects for each entity as well as for the sort *Users*, and where the interpretations of `String`, `Int`, `Real`, and `Boolean` are fixed to be the sets *Strings*, $\mathbb{Z}$, $\mathbb{R}$, and $\{\mathit{true}, \mathit{false}\}$ respectively.

  The entity sorts consist of objects that can be thought of as tuples, containing an object identifier and fields for each attribute. The attribute fields contain the object identifier of the referenced object

(in case this object is of an entity sort) or a value of one of the primitive types.

- The set of actions A is defined by (cf. Figure 8):

$$\begin{aligned} A = {} & \mathit{EntityCreateActions} \cup \mathit{EntityDeleteActions} \cup \\ & \mathit{MethodActions} \cup \\ & \mathit{AttributeReadActions} \cup \mathit{AttributeUpdateActions} \cup \\ & \mathit{AssociationEndReadActions} \cup \mathit{AssociationEndAddActions} \cup \\ & \mathit{AssociationEndRemoveActions} \ , \end{aligned}$$

  where, for example, AttributeUpdateActions is defined by:

$$\mathit{AttributeUpdateActions} = \bigcup_{\{at \in \mathit{Attributes}\}} \{set_{at}\} \times Q_e \times Q_{at} \ .$$

  Here, Q_e and Q_{at} denote the universes of all possible instances of the type of the attribute's entity, and the type of the attribute respectively, e.g., the action $(set_{at}, e, v) \in \mathit{AttributeUpdateActions}$ denotes the action of setting the attribute at of the entity e to the value v. The other sets of actions are defined similarly.

- The transition relation $\delta \subseteq Q \times A \times Q$ defines the allowed transitions. The exact details of δ will depend on the intended semantics of the methods themselves. We will just give a few examples here to illustrate the main idea. For example, for $a \in \mathit{AttributeReadActions}$, $(q, a, q') \in \delta$ if and only if $q = q'$, i.e., reading an attribute's value does not change the system state. In contrast, setting an attribute value should be reflected in the system state: for $a = (set_{at}, e, v) \in \mathit{AttributeUpdateActions}$, $(q, a, q') \in \delta$ implies $q' \models get_{at}(e) = v$.

It is possible to complete this account and give a full semantics of ComponentUML, but this would take us too far afield. Any completion will meet the requirements put forth in Section 4.3 and have a well-defined behavioral semantics. Specifically, the combined transition system $\Delta_{AC} = (Q_{AC}, A_{AC}, \delta_{AC})$ is defined by adding Σ_{RBAC}-structures to the system states in Q, extending δ to $Q_{AC} \times A_{AC} \times Q_{AC}$, and removing forbidden transitions. Hence, δ_{AC} will only contain those transitions that are allowed according to the SecureUML semantics.

5.4 Modeling the Authorization Policy

We now use the combined language to formalize the security policy given in Section 2.1. We do this by adding permissions to the entity

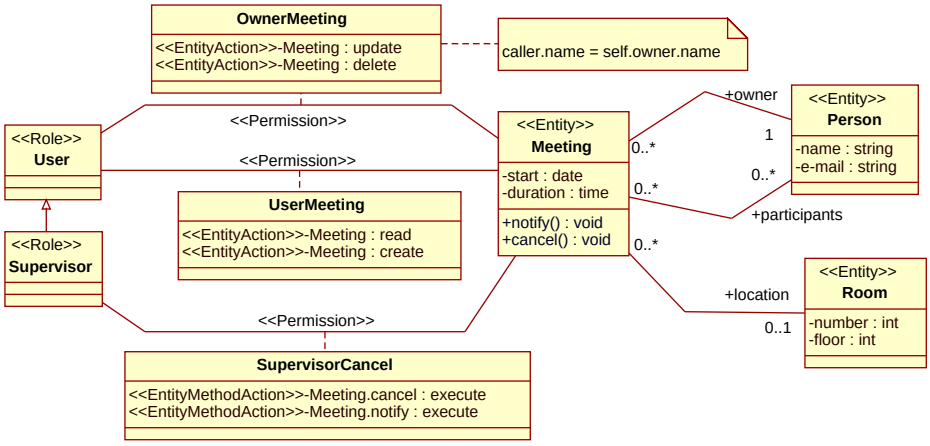

Figure 9. Scheduler Example with Authorization Policy

model of the scheduler application that formalize the three policy requirements. As these permissions associate roles with actions, we also employ the roles User and Supervisor, which we introduced in Section 4.2.

The first requirement states that any user may create and read meeting data. We formalize this by the permission UserMeeting in Figure 9, which grants the role User the right to perform *create* and *read* actions on the entity Meeting.

We formalize the second requirement with the permission OwnerMeeting, which states that a meeting may only be altered or deleted by its owner. This permission grants the role User the privilege to perform *update* and *delete* actions on a Meeting. Additionally, we restrict this permission with the authorization constraint `caller.name = self.owner.name`, which states that the name of a caller must be equal to the name of the owner of the meeting instance. Due to the definition of the action update (cf. Table 2), this permission must hold for every change of the value of the attributes or association ends of the meeting entity as well as for invocations of the methods notify or cancel.

Finally, we formalize the third requirement with the permission SupervisorCancel. This gives a supervisor the permission to cancel any meeting, i.e., the right to execute the methods cancel and notify.

5.5 Examples of Access Control Decisions

We now illustrate the semantics by analyzing several access control decisions in the context of Figure 9. We assume that we have three

users, Alice, Bob, and Jack, and that Bob is assigned the role `User` whereas Alice is assigned the role `Supervisor`. Here we assume that our dialect has the *default behavior* "access allowed" and we directly apply the semantics of SecureUML to the policy given in the previous section. The corresponding Σ_{RBAC}-structure $\mathfrak{S}_{RBAC}$ is[5]

$$\begin{aligned}
Users = Subjects =& \{\text{Alice}, \text{Bob}, \text{Jack}\} \\
Roles =& \{\texttt{User}, \texttt{Supervisor}\} \\
Permissions =& \{\texttt{OwnerMeeting}, \texttt{SupervisorCancel}, \dots\} \\
AtomicActions =& \{\texttt{Meeting::cancel.execute}, \dots\} \\
Actions =& AtomicActions \cup \{\texttt{Meeting.update}, \dots\} \\
UA =& \{(\text{Bob}, \texttt{User}), (\text{Alice}, \texttt{Supervisor})\} \\
PA =& \{(\texttt{User}, \texttt{OwnerMeeting}), \\
& \quad (\texttt{Supervisor}, \texttt{SupervisorCancel}), \dots\} \\
AA =& \{(\texttt{SupervisorCancel}, \texttt{Meeting::cancel.execute}), \\
& \quad (\texttt{OwnerMeeting}, \texttt{Meeting.update}), \dots\} \\
\geq_{Roles} =& \{(\texttt{Supervisor}, \texttt{User}), (\texttt{Supervisor}, \texttt{Supervisor}), \\
& \quad (\texttt{User}, \texttt{User})\} \\
\geq_{Actions} =& \{(\texttt{Meeting.update}, \texttt{Meeting::cancel.execute}), \\
& \quad \dots\} \ ,
\end{aligned}$$

and the signature Σ_{ST}, derived from the system model, is

$$\begin{aligned}
S &= \{Meetings, Persons, Rooms\} \cup \{\texttt{String}, \texttt{Int}, \texttt{Real}, \texttt{Bool}\} \\
F &= \{self_{\,Meetings}, \dots, MeetingOwner, PersonName\} \\
P &= \{MeetingLocation, MeetingParticipants, \dots\} \ .
\end{aligned}$$

The constant symbol $self_{\,Meetings}$ of sort *Meetings* denotes the currently accessed meeting. The function symbols

$$\begin{aligned}
MeetingOwner &: Meetings \to Persons \\
PersonName &: Persons \to String
\end{aligned}$$

represent the association end owner of the entity type Meeting and the attribute name of a person.

Now suppose that Alice wants to cancel a meeting entry owned by Jack. Suppose further that the system state is given by the first-order

structure $\mathfrak{S}_{st}$ over Σ_{ST}, where

$$\begin{aligned}
\texttt{caller}^{\mathfrak{S}_{st}} &= \text{Alice} \\
Meetings^{\mathfrak{S}_{st}} &= \{meeting_{\text{Jack}}\} \\
Persons^{\mathfrak{S}_{st}} &= \{alice, bob, jack\} \\
self_{Meetings}^{\mathfrak{S}_{st}} &= meeting_{\text{Jack}} \\
MeetingOwner^{\mathfrak{S}_{st}} &= \{(meeting_{\text{Jack}}, jack)\} \\
PersonName^{\mathfrak{S}_{st}} &= \{(alice, \texttt{"Alice"}), (bob, \texttt{"Bob"}), (jack, \texttt{"Jack"})\} \\
UserName^{\mathfrak{S}_{st}} &= \{(\text{Alice}, \texttt{"Alice"}, (\text{Bob}, \texttt{"Bob"}), (\text{Jack}, \texttt{"Jack"})\}.
\end{aligned}$$

The formula that must be satisfied by the structure $\mathfrak{S}_{AC} = \langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{st} \rangle$ in order to grant Alice access is built according to the definition (2), given in Section 4.3:

$$\phi_{AC}(u, a) = \bigvee_{p \in Permissions} \phi_{User}(u, p) \wedge \phi_{Action}(p, a) \wedge \phi_{ST}^{p}(u) \ .$$

As can be seen in Figure 9, Alice has the permission SupervisorCancel for the action `Meeting::cancel.execute`. However, the method `cancel()` of the entity Meeting is a method with side-effects. Therefore, the composite action `Meeting.update` includes the action `Meeting::cancel.execute`. Because the role Supervisor inherits permissions from the role User, Alice also has the permission OwnerMeeting for the action `Meeting::cancel.execute`. No other permissions for this action exist. Hence, the formula

$$\phi_{User}(\text{Alice}, p) \wedge \phi_{Action}(p, \texttt{Meeting::cancel.execute})$$

is only true for these permissions. The constraint expression

```
caller.name = self.owner.name
```

on the permission OwnerMeeting is translated into the formula

$$UserName(\texttt{caller}) = PersonName(MeetingOwner(self_{Meetings}())) \ ,$$

and the formula for the permission SupervisorCancel is *true*. For all other permissions p, the formula $\phi_{User}(u, p) \wedge \phi_{Action}(p, a)$ is false. Therefore the access decision is equivalent to

$$\begin{aligned}
&\mathfrak{S}_{AC} \models true \vee \\
&\quad UserName(\texttt{caller}) = PersonName(MeetingOwner(self_{Meetings}())) \, ,
\end{aligned}$$

which is satisfied.

Alternatively, suppose that Bob tries to perform this action. The corresponding structure $\mathfrak{S}'_{AC}$ differs from $\mathfrak{S}_{AC}$ by the interpretation of the constant symbol `caller`, which now refers to "Bob". Bob only has the permission OwnerMeeting for this action. Hence,

$$\mathfrak{S}'_{AC} \models \mathit{UserName}(\texttt{caller}) = \mathit{PersonName}(\mathit{MeetingOwner}(\mathit{self}_{\mathit{Meetings}}()))$$

is required for access. Since Jack (not Bob) is the owner of this meeting, this constraint is not satisfied and access is denied.

6. Generating an EJB System

We now show how ComponentUML models can be transformed into executable EJB systems with configured access control infrastructures. First, we outline the basic generation rules for EJB systems. Afterwards, we present the rules for transforming SecureUML elements into EJB access control information. The generation of users, roles, and user assignments is straightforward in EJB: for each user, role, and user assignment, we generate a corresponding element in the deployment descriptor. We therefore omit these details and focus here on the parts of the infrastructure responsible for enforcing permissions and authorization constraints.

6.1 Basic Generation Rules for EJB

Generation rules are defined for entities, their attributes, methods, and association ends. The result of the transformation is a source code fragment in the concrete syntax of the EJB platform, either Java source code or XML deployment descriptors.

An Entity is transformed to a complete EJB component of type *entity bean* with all necessary interfaces and an implementation class. Additionally, a factory method create for creating new component instances is generated. The component itself is defined by an entry in the deployment descriptor of type entity as shown by the following XML fragment.

```
<entity>
  <ejb-name>Meeting</ejb-name>
  <local-home>scheduler.MeetingHome</local-home>
  <local>scheduler.Meeting</local>
  <ejb-class>scheduler.MeetingBean</ejb-class>
  ...
</entity>
```

A Method is transformed to a method declaration in the component interface of the respective entity bean and a method stub in the corre-

sponding bean implementation class. The following shows the stub for the method cancel of the entity Meeting.

```
void cancel(){ }
```

For each Attribute, access methods for reading and writing the attribute value are generated along with persistency information that is used by the application server to determine how to store this value in a database. The declarations of the access methods for the attribute duration of the entity Meeting are shown in the following Java code fragment.

```
int getDuration();
void setDuration(int duration);
```

Elements of type AssociationEnd are handled analogously to attributes. Access methods are generated for reading the collection of associated objects and for adding objects to, or deleting them from, the collection. Furthermore, persistency information for storing the association-end data in a database is generated. The following code fragment shows the declarations of the access methods for the association end participants of the entity Meeting.

```
Collection getParticipants();
void addParticipant(Participant participant);
void removeParticipant(Participant participant);
```

6.2 Generating Access Control Infrastructures

We define generation rules that translate a security design model into an EJB security infrastructure based on declarative and programmatic access control. Each permission is translated into an equivalent XML element of type `method-permission`, used in the deployment descriptor for the declarative access control of EJB. The resulting access control configuration enforces the static part of an access control policy, without considering the authorization constraints. Programmatic access control is used to enforce the authorization constraints. For each method that is restricted by at least one permission with an authorization constraint, an assertion is generated and placed at the start of the method body.

Note that since the default behavior of both the SecureUML dialect for ComponentUML and the EJB access control monitor is "access allowed", we need not consider actions without permissions during generation.

Generating Permissions As explained in Section 2.5, a method permission element names a set of roles and the set of EJB methods that the members of the roles may execute. Generating a method permission can therefore be split into two parts: generating a set of roles and assigning methods to them.

rule #	resource type	action	EJB methods
1	Entity	create	automatically generated factory methods
2	Entity	delete	delete methods
3	Method	execute	corresponding method
4	Attribute	read	get-method of the attribute
5	Attribute	update	set-method of the attribute
6	AssociationEnd	read	get-method of the association end
7	AssociationEnd	update	add- and remove-method of the association end

Table 4. Atomic Action to Method Mapping for EJB

Since EJB does not support role hierarchies, both the roles directly connected to permissions in the model, as well as their subroles, are needed for generation. First, the set of roles directly connected to a permission is determined using the association PermissionAssignment of the SecureUML metamodel. Then, for every role in this set, all of its subroles (under the transitive closure of the relation defined by the association RoleHierarchy) are added to the role set. Finally, for each role in the resulting set, one `role-name` element is generated. Applying this generation procedure to the permission OwnerMeeting in our example results in the following two role references.

```
<role-name>User</role-name>
<role-name>Supervisor</role-name>
```

The set of `method` elements that is generated for each permission is computed similarly. First, for each permission, we determine the set of actions directly referenced by the permission using the association ActionAssignment. Then, for every action in this set, all of its subordinated actions (under the reflexive closure of the relation defined by the association ActionHierarchy) are added to the action set. Finally, for each atomic action in the resulting set, `method` elements for the corresponding EJB methods are generated. The correspondence between atomic actions and EJB methods is given in Table 4. Note that an atomic action may map to several EJB methods and therefore several `method` entries may need to be generated.

We illustrate this process for the permission UserMeeting, which references the actions `Meeting.create` and `Meeting.read`. The resulting set of atomic actions for this permission is

{`Meeting.create`, `Meeting::start.read`, `Meeting::duration.read`,
`Meeting::owner.read`, `Meeting::location.read`, `Meeting::participants.read`} ,

where "`::`" is standard object-oriented notation, which is used here to reference the attributes and association ends of the entity `Meeting`. The

action create of the entity Meeting remains in the set, whereas the action read is replaced by the corresponding actions for reading the attributes and the association ends of the entity Meeting. The mapping rules 1, 4, and 6 given in Table 4 are applied, which results in a set of six methods: the method create, the read-methods of the attributes start and duration, and the read-methods of the association ends owner, participants, and location. The XML code generated is as follows:

```
<method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>create</method-name>
  <method-params/>
</method>
<method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>
    getStart
  </method-name>
  <method-params/>
</method>
<method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>
    getDuration
  </method-name>
  <method-params/>
</method>
```

```
<method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>getOwner</method-name>
  <method-params/>
</method>
<method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>
    getLocation
  </method-name>
  <method-params/>
</method>
 <method>
  <ejb-name>Meeting</ejb-name>
  <method-intf>Local</method-intf>
  <method-name>
     getParticipants
  </method-name>
  <method-params/>
</method>
```

Generating Assertions While the generation of an assertion for each OCL constraint is a simple matter, this task is complicated by the fact that a method may have multiple (alternative) permissions, associated with different constraints and roles, where the roles in turn may be associated with subroles. Below we describe how we account for this when generating assertions.

First, given a method m, the atomic action a corresponding to the method is determined using Table 4. For example, the action corresponding to the EJB method `Meeting::cancel` is the action *execute* of the method cancel of the entity Meeting in the model. Then, using this action a, the set of permissions ActionPermissions(a) that affect the execution of the method m is determined as follows: a permission is included if it is assigned to a by the association ActionAssignment or one of the super-actions of a (under the reflexive closure of the relation defined by the association ActionHierarchy). Next, for each permission p in the resulting set ActionPermissions(a), the set PR(p) of roles assigned to p is determined, again taking into account the hierarchy on roles in the same way as in the previous section. Finally, based on this information, an assertion is generated of the form

$$\texttt{if (!(} \bigvee_{p \in ActionPermissions(a)} \Big(\big(\bigvee_{r \in PR(p)} UserRole(r) \big) \wedge Constraint(p) \Big) \texttt{))} \quad \texttt{throw new AccessControlException("Access denied.");} \; . \tag{3}$$

This scheme is similar to Equation (2) in Section 4.3, which defines $\phi_{\mathrm{AC}}(u, a)$, as each permission represents an (alternative) authorization to execute an action. However, because the permission assignments and action assignments are known at compile time, this information is used to simplify the assertion. Instead of considering all permissions, we only consider permissions that refer to the action in question by calculating the set $ActionPermissions(a)$. This has the effect that the equivalent of $\phi_{Action}(p, a)$ in Equation (2) can be omitted. Similarly, the equivalent of ϕ_{User} is simplified by only considering roles that have one of these permission, which is done by calculating the sets $PR(p)$. If a constraint is assigned to a permission, it is evaluated afterwards. Access denial is signaled to the caller by throwing an exception.

As an example, the following assertion is generated for the method `Meeting::cancel`.

```
if (!(ctxt.isCallerInRole("Supervisor")/* SupervisorCancel */
      || (ctxt.isCallerInRole("User")
          || ctxt.isCallerInRole("Supervisor"))
      && ctxt.getCallerPrincipal.getName().equals(
         getOwner().getName()))) /* OwnerMeeting */
    throw new AccessControlException("Access denied.");
```

Observe that the role assignment check $UserRole(r)$ is translated into a Java expression of the form `ctxt.isCallerInRole(<roleName>)`. The variable `ctxt` references an object that is used in EJB to communicate with the execution environment of a component. Here, the context object is used to check the role assignment of the current caller.

An authorization constraint, defined in OCL, is translated to an equivalent Java expression. The symbol caller is translated into the expression `ctxt.getCallerPrincipal.getName()`. Access to methods, attributes, and association ends respects the rules that are applied to generate the respective counterparts of these elements, given in Section 6.1. For example, access to the value of an attribute name is translated to a call of the corresponding read method `getName`. The OCL equality operator is translated into the Java method `equals` for objects or into Java's equality operator for primitive types.

7. ControllerUML

To demonstrate the general applicability of our approach, we now present a second design modeling language. This language, which we

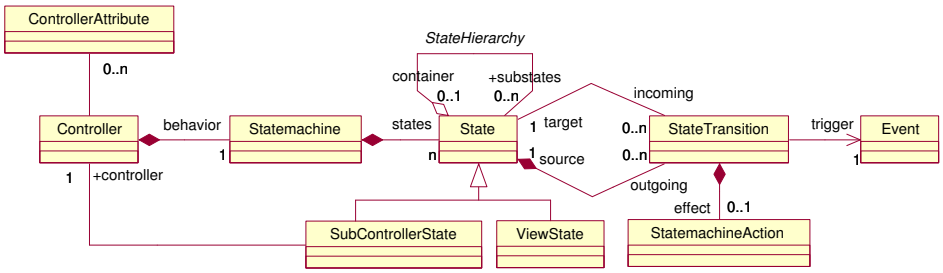

Figure 10. Metamodel of ControllerUML

call *ControllerUML*, is based on state machines.[6] We will show how ControllerUML can be integrated with SecureUML and used to model secure controllers for multi-tier applications, and how access control infrastructures can be generated from such controller models.

A well-established pattern for developing multi-tier applications is the Model-View-Controller pattern [Krasner and Pope, 1988]. In this pattern, a controller is responsible for managing the control flow of the application and the data flow between the persistence tier (model) and the visualization tier (view). The behavior of the controller can be formalized by using event-driven state machines and the modeling language ControllerUML utilizes UML state machines for this purpose.

The abstract syntax of ControllerUML is defined by the metamodel shown in Figure 10. Each Controller possesses a Statemachine that describes its behavior in terms of States, StateTransitions, Events, and StatemachineActions. A State may contain other states, formalized by the association StateHierarchy, and a transition between two states is defined by a StateTransition, which is triggered by the event referenced by the association end trigger. A state machine action specifies an executable statement that is performed on entities of the application model. ViewState and SubControllerState are subclasses of State. A ViewState is a state where the application interacts with humans by way of view elements like dialogs or input forms. The view elements generate events in response to user actions, e.g., clicking a mouse button, which are processed by the controller's state machine. A SubControllerState references another controller using the association end controller. The referenced controller takes over the application's control flow when the referencing SubControllerState is activated. This supports the modular specification of controllers.

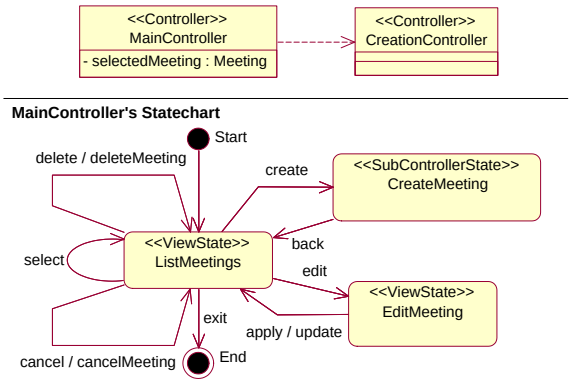

Figure 11. Controllers for the Scheduling Application

The notation of ControllerUML uses primitives from UML class diagrams and statecharts. An example of a ControllerUML model is shown in Figure 11. A Controller is represented by a UML class with the stereotype «Controller». The behavior of the controller is defined by a state machine that is associated with this class. States, transitions, events, and actions are represented by their counterparts in the UML metamodel. Transitions are labeled with a string, containing a triggering event and an action to be executed during state transition, separated by a slash. We use events to name transitions in our explanations. View states and subcontroller states are labeled by the stereotypes «ViewState» and «SubControllerState», respectively.

Figure 11 shows the design model for an interactive application that formalizes the scheduler workflow presented in Section 2.2. The controller class MainController is the top-level controller of the application and CreationController controls the creation of new meetings (details are omitted here to save space). The state machine of MainController is similar to that of Figure 2. Note that the selected meeting is stored in the attribute selectedMeeting of the controller object. Also, the reference from the subcontroller state CreateMeeting to the controller CreationController is not visible in the diagram. This information is stored in a tagged value of the subcontroller state.

7.1 Extending the Abstract Syntax

There are various ways to introduce access control into a process-oriented modeling language like ControllerUML. For example, one can choose whether entering states or making transitions (or both) are pro-

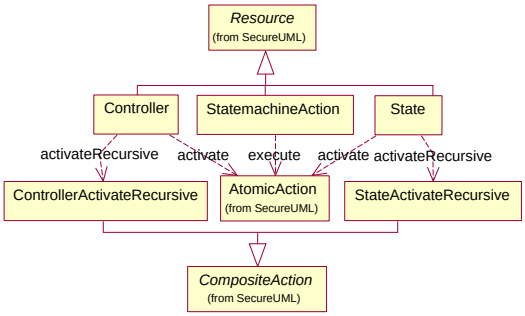

Figure 12. Resource Model of ControllerUML

tected. Each choice results in the definition of a different dialect for integrating ControllerUML with SecureUML. Here we shall proceed by focusing on the structural aspects of statecharts, which are described by the classes of the metamodel (Figure 10) and the relations between them. We identify the types Controller, State, and StatemachineAction as the resource types in our language since their execution or activation can be sensibly protected by checkpoints in the generated code. Figure 12 shows this identification and also defines the composite actions for the dialect and the assignment of actions to resource types.

The resource type StatemachineAction offers the atomic action *execute* and a state has the actions *activate* and *activateRecursive*. The action *activateRecursive* on a state is composed of the actions *activate* on the state, *execute* on all state machine actions of the outgoing transitions of the state, and the actions *activateRecursive* on all substates of the state. The corresponding OCL definition is as follows:

```
context StateActivateRecursive inv:
subordinatedActions =
  resource.actions->select(name = "activate")
  ->union(resource.outgoing->select(effect<>None).effect
          .actions->select(name = "execute"))
  ->union(resource.substates.actions
          ->select(name = "activateRecursive"))) .
```

This expression is built using the vocabulary defined by the ControllerUML metamodel shown in Figure 10 and the dialect definition given in Figure 12. The third line accesses the resource that the action belongs to (always a state) and selects the action with the name “activate”. The next line queries all outgoing transitions on the state and selects those transitions with an assigned state machine action (association end ef-

stereotype	resource type	naming convention
ControllerAction	Controller	empty string
StateAction	State	state name
ActionAction	StatemachineAction	state name + "." + event name

Table 5. Action Reference Types for ControllerUML

fect). Afterwards, for each state machine action, its (SecureUML) actions with the name "execute" is selected. The last line selects all actions with the name "activateRecursive" on all substates of the state to which the action of type StateActivateRecursive belongs.

A controller possesses the actions *activate* and *activateRecursive*. The latter is a composite action that includes the action *activate* on the controller and the action *activateRecursive* for all of its states. Due to the definition of *activateRecursive* on states, this (transitively) includes all substates and all actions of the state machine.

7.2 Extending the Notation

First, we merge the notation of both languages. Afterwards, we define well-formedness rules on SecureUML primitives that restrict which kinds of combined expressions are possible, i.e., we restrict how SecureUML primitives can refer to ControllerUML elements representing protected resources. For example, the scope of a permission is restricted to the UML classes with the stereotype «Controller». Finally, we define the action reference types for controllers, states, and state machine actions, as shown in Table 5.

7.3 Extending the Semantics

We first define the semantics of ControllerUML in terms of a labeled transition system over a fixed first-order signature (cf. Section 4.3). Intuitively, every Controller defines a sort in the first-order signature and, in addition, we have a sort of states. Also, every atomic action defined in the SecureUML dialect as well as every state-transition in the ControllerUML model defines an action of the labeled transition system.

More precisely, given a model in the ControllerUML language, the corresponding signature $\Sigma_{ST} = (\mathcal{S}_{ST}, \mathcal{F}_{ST}, \mathcal{P}_{ST})$ is defined as follows:

- Each Controller c gives rise to two sorts C_c and S_c in $\mathcal{S}_{ST}$. C_c is the sort of the controller c, where the elements of sort C_c represent the instances of the controller c. Each user interacting with the system gives rise to such an instance. S_c is the sort of the states of

the controller c, where each state of the state machine describing the behavior of the controller c gives rise to an element of sort S_c. Additionally, $\mathcal{S}_{ST}$ contains the sorts Users, String, Int, Real, and Boolean:

$$\mathcal{S}_{ST} = \{S_c \mid c \text{ is a controller}\} \cup \{C_c \mid c \text{ is a controller}\} \cup \{\mathit{Users}, \texttt{String}, \texttt{Int}, \texttt{Real}, \texttt{Boolean}\} \ .$$

- Function symbols are defined similarly to ComponentUML. However, controllers in ControllerUML only have attributes, but not methods. Therefore, each controller attribute at gives rise to a function symbol get_{at} in $\mathcal{F}_{ST}$ (the "get-method") of type $s \to v$, where s is the sort of the controller, and v is the sort of the attribute's type:

$$\mathcal{F}_{ST} = \{get_{at} \mid at \text{ is a controller attribute}\} \cup \{self_c \mid c \text{ is a controller}\} \cup \{\mathit{caller}, \mathit{UserName}\} \ .$$

  The initial and current states of a controller's state machine are denoted by the implicit (in the sense that every controller will have them) controller attributes `initialState` and `currentState` of type S_c. The initial state of a controller denotes the state that is active when the state machine starts after the controller is created, and the current state denotes the currently active state. Whereas the attributes `initialState` and `currentState` are of type S_c, other controller attributes denote application-specific data attached to the controller and can have the types `String`, `Int`, `Real`, and `Boolean`. Additionally, it is possible to combine ControllerUML with a more data-oriented modeling language (like ComponentUML). Then one can use controller attributes with types provided by the data-modeling language. For example, in Figure 13 in the MainController, we refer to the entity Meeting of the ComponentUML model.

- Since there are no predicate symbols,

$$\mathcal{P}_{ST} = \emptyset \ .$$

The transition system $\Delta = (Q, A, \delta)$ is defined as follows:

- Q is the universe of all possible states, which is just the set of all first-order structures over the signature Σ_{ST} with finitely many elements for each controller sort as well as for the sort *Users*, where the interpretations of `String`, `Int`, `Real`, `Boolean`, and S_c are fixed

to be the sets *Strings*, $\mathbb{Z}$, $\mathbb{R}$, $\{true, false\}$, and the set of states of the controller c respectively.

- The set of actions A is defined by:

$$A = ControllerActivateActions \cup StateActivateActions \cup SMActionExecuteActions \cup StateTransitions \ .$$

  This means that all atomic actions (cf. Figure 12) as well as all state transitions are actions of the transition system.

- The transition relation $\delta \subseteq Q \times A \times Q$ defines the allowed transitions. For example, one requires that for each transition $s_1 \xrightarrow{a} s_2$ in the model there are corresponding tuples (s_{old}, a, s_{new}) in δ, where the current state of the controller (i.e., the attribute `currentState`) is s_1 in s_{old} and is s_2 in s_{new}. For the purposes of this paper, it does not matter which particular semantics is used, e.g., one of the many semantics for state-chart like languages ([von der Beeck, 1994] lists about 20 of them).

Having defined the semantics of ComponentUML in this way, we combine it with the semantics of SecureUML as described in Section 4.3. That is, the new transition system $\Delta_{AC} = (Q_{AC}, A_{AC}, \delta_{AC})$ is defined by adding Σ_{RBAC}-structures to the system states in Q, extending δ to $Q_{AC} \times A_{AC} \times Q_{AC}$, and removing the forbidden transitions from the result. Hence, δ_{AC} will only contain those transitions that are allowed according to the SecureUML semantics.

7.4 Formalizing the Authorization Policy

We now return to our scheduling application model and extend it with a formalization of the security policy given in Section 2.1. In doing so, we use the role model introduced in Section 4.2.

As Figure 13 shows, we use two permissions to formalize the first requirement that all users are allowed to create and to read all meetings. The permission UserMain grants the role User the right to activate the controller MainController and the states ListMeetings and CreateMeeting. The permission UserCreation grants the role User the privilege to activate the CreationController including the right to activate all of its states and to execute all of its actions.

The second requirement states that only the owner of a meeting entry is allowed to change or delete it. We formalize this by the permission OwnerMeeting, which grants the role User the right to execute the actions on the outgoing transitions delete and cancel of the state ListMeetings and

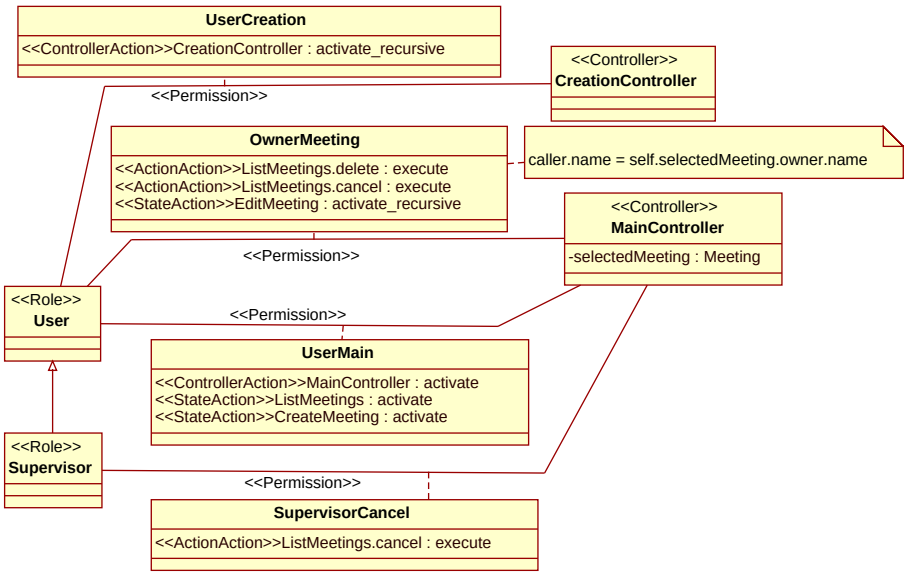

Figure 13. Policy for Scheduling Application

the right to activate the state EditMeeting. This permission is restricted by the ownership constraint attached to it.

Finally, only supervisors are allowed to cancel any meeting. Therefore, the permission SupervisorCancel grants this role the unrestricted right to execute the action cancelMeeting on the transition cancel.

7.5 Transformation to Web Applications

In this section, we describe a transformation function that constructs secure web applications from ControllerUML models. As a starting point, we assume the existence of a transformation function that translates UML classes and state machines into controller classes for web applications, which can be executed in a Java Servlet environment (see Section 2.5). We describe here how we extend such a function to generate security infrastructures from SecureUML models.

The Java Servlet architecture 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 advanced 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 declarative authorization mechanism only provides protection for the dispatcher. To overcome this weakness, we generate access control infrastructures that exploit the programmatic access control mechanism that servlets provide, where the role assignments of a user can be retrieved by any servlet.

Our transformation function is an extension of an existing generator provided by the MDA-tool ArcStyler [Hubert, 2001], which converts UML classes and state machines into controller classes. Each controller is equipped with methods for activating the controller, performing state transitions, activating the states of the controller, and executing actions on transitions.

We augment the existing transformation function by generation rules that operate on the abstract syntax of SecureUML and add Java assertions to the methods for *process activation*, *state activation*, and *action execution* of a controller class. First, the set $ActionPermissions(a)$, which contains all permissions affecting the execution of an action, is determined as described in Section 6.2. Afterwards, an assertion is generated of the form:

$$\texttt{if (!(} \bigvee_{p \in ActionPermissions(a)} \Big(\big(\bigvee_{r \in PR(p)} UserRole(r) \big) \wedge Constraint(p) \Big) \texttt{))} \quad \texttt{c.forward("/unauthorized.jsp");} \tag{4}$$

The rule that generates this assertion has a structure similar to rule 3 in Section 6.2, which is used to generate assertions in the stubs of EJB components. However, instead of throwing an exception when access is denied, a request to a controller is forwarded to an error page by the term `c.forward("/unauthorized.jsp")`. Additionally, the functions used to obtain security information differ between EJB and Java Servlet. For example, the following assertion is generated for the execution of the action cancel on the state ListMeetings.

```
if (!(request.isUserInRole("Supervisor") /*SupervisorCancel*/
      || (request.isUserInRole("User")
          || request.isUserInRole("Supervisor"))
      && getSelectedMeeting().getOwner().getName().equals(
         request.getRemoteUser())))
    c.forward("/unauthorized.jsp");
```

The role check is performed using the method *isUserInRole*() on the request object and each constraint is translated into a Java expression that accesses the attributes and side-effect free methods of the controller. The symbol caller is translated into a call to *getRemoteUser*() on the `request` object.

8. Conclusion

8.1 Evaluation

We have evaluated the ideas presented in this paper in an extensive case study: the model-driven development of the J2EE "Pet Store" application. Pet Store is a prototypical e-commerce application designed to demonstrate the use of the J2EE platform. It features web front-ends for shopping, administration, and order processing. The application model consists of 30 components and several front-end controllers. We have extended this model with an access control policy formalizing the principle of least privileges, where a user is given only those access rights that are necessary to perform a job. The modeled policy comprises six roles and 60 permissions, 15 of which are restricted by authorization constraints. The corresponding infrastructure is generated automatically and consists of roughly 5,000 lines of XML (overall application: 13,000) and 2,000 lines of Java source code (overall application: 20,000).

This large expansion is due to the high abstraction level provided by the modeling language. For example, we can grant a role read access to an entity, whereas EJB only supports permissions for whole components or single methods. Therefore, a modeled permission to read the state of a component may require the generation of many method permissions, e.g., for the get-methods of all attributes. Clearly, this amount of information cannot be managed at the source code level. The low abstraction level provided by the access control mechanisms of today's middleware platforms often forces developers to take shortcuts and make compromises when implementing access control. For example, roles are assigned full access privileges even where they only require read access. As our experience shows, Model Driven Security can not only help to ease the transition from security requirements to secure applications, it also plays an important role in helping system designers to formalize and meet exact application requirements.

8.2 Related Work

Various extensions to the core RBAC model have been presented in the literature, e.g., [Jaeger, 1999; Ahn and Sandhu, 1999; Ahn and Sandhu, 2000; Ahn and Shin, 2001]. These use constraints on role assignments to express different kinds of high-level organizational policies, like separation of duty. In contrast, SecureUML extends RBAC with constraints on system states associated with a design model.

Jürjens [Jürjens, 2001; Jürjens, 2002] proposed an approach to developing secure systems using an extension of UML called UMLsec. Using

UMLsec, one can annotate UML models with formally specified security requirements, like confidentiality or secure information flow. In contrast, our work focuses on a semantic basis for annotating UML models given by class or statechart diagrams with access control policies, where the semantics provides a foundation for generating implementations and for analyzing these policies.

Probably the most closely related work is the Ponder Specification Language [Damianou, 2002], which supports the formalization of authorization policies where rules specify which actions each subject can perform on given targets. As in our work, Ponder supports the organization of privileges in an RBAC-like way and allows rules to be restricted by conditions expressed in a subset of OCL. Moreover, Ponder policies can be directly interpreted and enforced by a policy management platform.

There are, however, important differences. The possible actions on targets are defined in Ponder by the target's visible interface methods. Hence, the granularity of access control in Ponder is at the level of methods, whereas in our approach higher-level actions can be defined using action hierarchies. Moreover, Ponder's authorization rules refer to a hierarchy of domains in which the subjects and targets of an application are stored. In contrast, our approach integrates the security modeling language with the design modeling language, providing a joint vocabulary for building combined models. In our view, the overall security of systems benefits by building such *security design models*, which tightly integrate security policies with system design models during system design, and using these as a basis for subsequent development.

8.3 Future Work

There are a number of promising directions for future work. To begin with, the languages we have presented constitute representative examples of security and design modeling languages. There are many questions remaining on how to design such languages and how to specialize them for particular modeling domains. On the security modeling side, one could enrich SecureUML with primitives for modeling other security aspects, like digital signatures or auditing. On the design modeling side, one could explore other design modeling languages that support modeling different views of systems at different levels of abstraction. What is attractive here is that our use of dialects to join languages provides a way of decomposing language design so that these problems can be tackled independently.

We believe that Model Driven Security has an important role to play not only in the design of systems but also in their analysis and certifica-

tion. Our semantics provides basis for formally verifying the transformation of models to code. Moreover, since our models are formal, we can ask questions about them and get well-defined answers, as the examples given in Section 5.5 suggest. More complex kinds of analysis should be possible too, which we will investigate in future work. Ideas here include calculating a symbolic description of those system states where an action is allowed, model checking statechart diagrams that combine dynamic behavior specifications with security policies, and verifying refinement or consistency relationships between different models.

Notes

1. SecureUML supports users, groups, and their role assignment. This can be used, e.g., to analyze the security-related behavior of an application. In general, user administration will not be performed using UML models, but rather using administration tools provided by the target platform at deployment time. Note too that the reader should not confuse the «Role» User with the SecureUML type «User».

2. For an overview of order-sorted signatures and algebras, see [Goguen and Meseguer, 1992].

3. Recall that authorization constraints are OCL formulas. A translation from OCL constraints to first-order formulas can be found in [Beckert et al., 2002].

4. Note that we are here combining a many-sorted signature and an order-sorted signature. This is sensible because a many-sorted signature is trivially order-sorted

5. We denote actions by the name of their resource and the name of the action type, separated by a dot.

6. To keep the account self-contained, we simplify state machines by omitting parallelism, actions on state entry and exit, and details on visualization elements.

References

Ahn, G.-J. and Sandhu, R. S. (1999). The RSL99 language for role-based separation of duty constraints. In *Proceedings of the 4th ACM Workshop on Role-based Access Control*, pages 43–54. ACM Press.

Ahn, G.-J. and Sandhu, R. S. (2000). Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226.

Ahn, G.-J. and Shin, M. E. (2001). Role-based authorization constraints specification using object constraint language. In *10th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2001)*, pages 157–162. IEEE Computer Society.

Akehurst, D. and Kent, S. (2002). A relational approach to defining transformations in a metamodel. 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 243–258. Springer Verlag.

Beckert, B., Keller, U., and Schmitt, P. H. (2002). Translating the Object Constraint Language into first-order predicate logic. In Autexier, S. and Mantel, H., editors, *Proceedings of the Second Verification Workshop: VERIFY'02*, volume 02-07 of *DIKU technical reports*, pages 113–123.

Damianou, N. (2002). *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College, University of London.

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

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

Goguen, J. A. and Meseguer, J. (1992). Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273.

Hubert, R. (2001). *Convergent Architecture: Building Model Driven J2EE Systems with UML*. John Wiley & Sons.

Hunter, J. (2001). *Java Servlet Programming, 2nd Edition*. O'Reilly & Associates.

Jaeger, T. (1999). On the increasing importance of constraints. In *Proceedings of 4th ACM Workshop on Role-based Access Control*, pages 33–42. ACM Press.

Jürjens, J. (2001). Towards development of secure systems using UMLsec. In Hussmann, H., editor, *Fundamental Approaches to Software Engineering (FASE/ETAPS 2001)*, number 2029 in LNCS, pages 187–200. Springer-Verlag.

Jürjens, J. (2002). UMLsec: Extending UML for secure systems development. In Jézéquel, J.-M., Hussmann, H., and Cook, S., editors, *UML 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer-Verlag.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Program.*, 1(3):26–49.

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

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

von der Beeck, M. (1994). A comparison of statechart variants. In Langmaack, H., de Roever, W.-P., and Vytopil, J., editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer Verlag.