Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report No. 414

Model Driven Security: from UML Models to Access Control Infrastructures

David Basin

Information Security Group, ETH Zentrum, CH-8092 Zurich, Switzerland
basin@inf.ethz.ch

Jürgen Doser   Torsten Lodderstedt

Institute for Computer Science, Albert-Ludwigs Universität Freiburg
{doser,tlodderstedt}@informatik.uni-freiburg.de

July 25, 2003

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 schema for constructing such languages that combines languages for modeling systems with languages for modeling security. We present different instances of this schema, which combine different UML modeling languages with a security modeling language for formalizing access control requirements. From models in these languages, we automatically generate security architectures for distributed applications, built from declarative and programmatic access control mechanisms. The modeling languages and generation process are semantically well-founded and are based on an extension of role-based access control. We have implemented this approach in a UML-based CASE tool and report on experiments.

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 system 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 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 [21]). 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

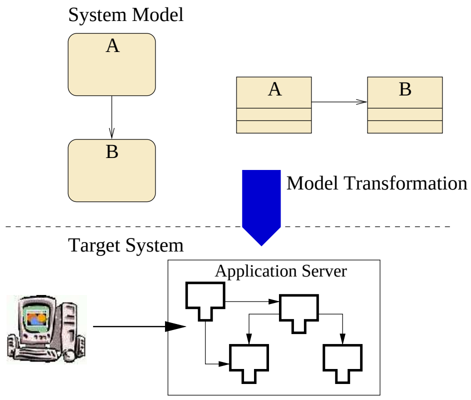

Figure 1: Model Driven Architecture

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) [11], which has been proposed as a means for supporting the software development process by employing a model-centric and generative approach. As Figure 1 suggests, the MDA approach has three parts: users create 1) 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. Examples include the generation of distributable components, including database access and transaction management, from class diagrams or the generation of controllers from statemachine models.

Our main contribution is to show how the Model Driven Architecture approach can be specialized to what we call *Model Driven Security*. As suggested by Figure 2, we specialize the three parts of MDA to model security requirements and generate security infrastructures. The most difficult part of this specialization concerns the first part and here we propose a general schema for integrating security requirements into system design models. 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) [10]. To support visual modeling, we embed this language within an extension of UML and hence we

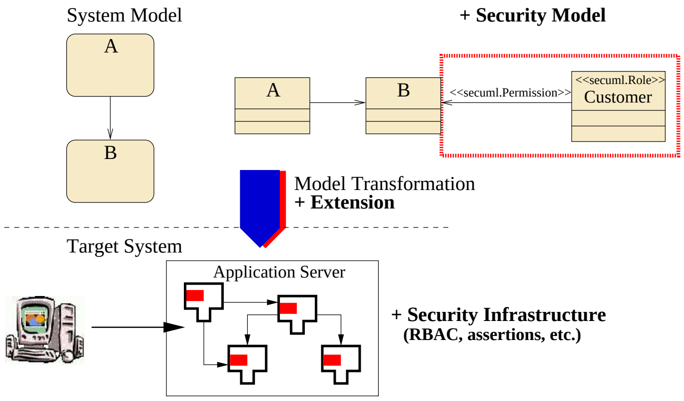

Figure 2: Model Driven Security

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 the 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 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 transformation functions that produce security infrastructures for distributed systems conforming to the Enterprise JavaBeans (EJB) standard or, alternatively, the Microsoft Enterprise Services for .NET. 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. These translations are based on a set-theoretic semantics for our SecureUML extensions that maps, essentially, our extensions into a relational structure with constraints. Given a SecureUML dialect, this semantic information is then translated into an access control infrastructure; for instance the relational structure is mapped into a configuration for declarative access control.

As a proof of concept, within the MDA-tool ArcStyler [14] 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 the final system. Moreover, it integrates security models with system design models and yields a new kind of model, *security design models*.

The model driven security approach that we propose offers additional advantages, which we briefly mention here. First, it naturally gives rise to models that are technology independent, reusable, and evolvable. As the technology specific details (e.g., application programming interfaces) are specified by the transformation functions, instead of by the models, architectures can be generated for (or evolved to) new technologies simply by changing the model transformation. We illustrate this by giving two different transformation functions that translate models defined in the SecureUML dialect for distributed object systems to either EJB or .NET technology. Second, by integrating security and system design models, it is possible to model and generate "security aware" applications that only present options to the user consistent with the formalized security policy. Finally, as our models have a formal semantics, it is possible to formally analyze both the models and the transformation process. Although we do not investigate this here, it should be possible to carry out automatic property checking of security design models to detect and correct design errors and even to verify the correctness of the model transformation process itself.

The remainder of the paper is organized as follows. Section 2 introduces the running example used in this paper and provides background on the Unified Modeling Language, Model Driven Architecture, Role-Based Access Control, and the security architectures of Enterprise JavaBeans, Enterprise Services for .NET, and Java Servlets. In Section 3 we give an overview of model driven security and in Section 4 we present the syntax and semantics of SecureUML, our security modeling language for access control. We show how to combine SecureUML with a design modeling language for component-oriented systems in Section 5, and how to generate Enterprise JavaBeans security infrastructures in Section 6 and .NET infrastructures in Section 7. To demonstrate the general applicability of our approach, in Section 8 we give a second example of integrating SecureUML with a design modeling language, this time for modeling the control flow of applications and generating a Java Servlet access control architecture. In Section 9 we review related work and in Section 10 we draw conclusions and discuss possible future work.

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.

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 also 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 (here informally given) security policy:

1. All users of the system 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. However, a supervisor can cancel any meeting.

We will later build models for this problem that will be automatically transformed into a system design for a multi-tier scheduling application along with a complete access control infrastructure.

2.2 The Unified Modeling Language

The Unified Modeling Language (UML) [21] is a widely used graphical language for modeling object-oriented systems. The language specification differentiates between an *abstract syntax*

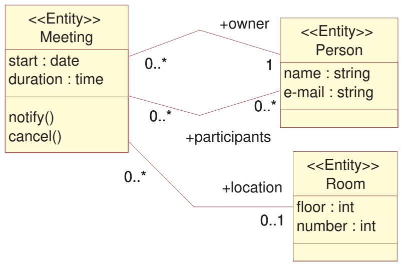

Figure 3: Scheduler Application Class Diagram

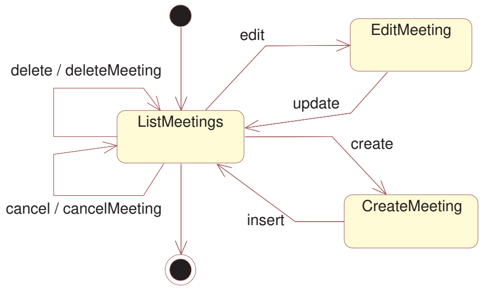

Figure 4: Scheduler Application Statechart

and a *notation* (also called *concrete syntax*). The abstract syntax defines the language primitives used to build models, whereas the notation defines the graphical presentation of these primitives as icons, strings, or geometric figures. UML supports the description of structural and behavioral aspects of a software system by 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, each class formalizing a set of objects with common services, properties, and behavior. Services are described by methods and properties by attributes and associations. Every class participating in an association is connected to the association by an association end, which may also specify the role name of the class and its cardinality in the association. The behavior of a class can be characterized by other UML elements such as a statemachine that is attached to the class.

Classes are depicted in class diagrams as shown in Figure 3. This diagram shows the structure of our scheduling application, supporting the planning of meetings, including room scheduling and the notification of participants about changes in the schedule. The model consist 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 planned room.

Statemachines describe the behavior of a system or a class in terms of the states of the system or class and the events that cause a transition between states. A statemachine is graphically represented by a statechart diagram. The rectangles and circles represent states and the arrows represent transitions. Transitions may be labeled with the name of the triggering event and (separated by a slash) the name of the action that is executed during state transition.

Figure 4 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 the Object Constraint Language (OCL), which is based on first-order logic. OCL expressions are used to formalize invariants for classes, preconditions and postconditions for methods, and guards for enabling transitions in a statemachine. As an example, we can add to the class `Meeting` in Figure 3 the following OCL constraint:

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

The constraint defines an invariant of the class `Meeting` stating that the owner of a meeting must be contained in the set of participants. Each OCL expression is evaluated in the context of an instance of a specific type and the reserved symbol `self` is used to refer to the instance. In our example, `self` represents an instance of the type `Meeting`. The attributes, association ends and methods of an instance can be accessed using "dot-notation". In our example expression, `participants` and `owner` denote the respective association ends of the meeting.

UML can also serve as foundation for building domain specific languages. *Stereotypes* are used to introduce new language primitives by subtyping core UML types, and *tagged values*, which are pairs of *tags* and *values*, represent new properties of these new language primitives. Model elements are assigned to such types by labeling them with the corresponding stereotype. Additional restrictions on the syntax of the domain specific language can be defined using OCL constraints. A set of such definitions constitutes a *UML profile*.

2.3 Model Driven Architecture

The Object Management Group (OMG) has proposed Model Driven Architecture (MDA) as an approach to specifying and developing applications based on platform-independent system models. Platform-specific details are incorporated in later stages of the development process by generating platform-specific models or even executable code directly from the platform-independent models.

UML plays a central role in MDA in two regards. First, the models created are given in domain specific specializations of UML. The specialization may be for a business domain like health care, a particular system aspect like security, or a particular technology like Enterprise JavaBeans. Second, UML is also used as a metamodeling language to define domain specific modeling languages themselves.

The definition of domain specific languages is at the heart of MDA. A common practice is to define a modeling language by giving an abstract syntax and a concrete syntax. The abstract syntax is usually defined by a metamodel in accordance with the OMG Meta-Object

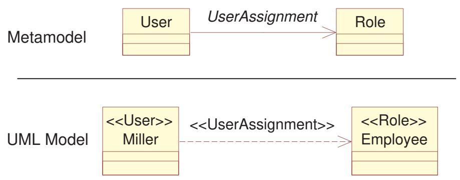

Figure 5: Example of a Domain Specific Modeling Language

Facility (MOF) [11]. As metamodeling falls outside the scope of this paper, we will not go into technical details here. However, one aspect of MOF that has a central role in our work is that it allows object-oriented concepts, like type abstraction and subtyping, to be used when defining languages. The notation used to describe a metamodel is typically the UML profile for MOF as defined in [20]. There, UML (meta)models are used to define the structure of the abstract syntax and OCL expressions are used to make the definition more precise by formulating additional requirements, called *well-formedness rules*, on the syntax.

The concrete syntax of a domain specific language is defined by a UML profile using stereotypes, tagged values, and constraints. MOF tools automatically translate models in the concrete syntax of a language into models in the abstract syntax for further processing.

As a simple example, Figure 5 shows parts of a definition of a language for describing access control. Above the line is the metamodel declaring the language primitives User and Role and the association UserAssignment, which represents the assignment of users to roles. An example of the concrete syntax is given below the line. There, users are represented by UML classes with the stereotype «User», roles as UML classes with the stereotype «Role», and an assignment of a user to a role is expressed by a UML dependency with the stereotype «UserAssignment».

Separating the abstract syntax of a language from its UML-based concrete syntax allows us to use UML CASE tools for modeling while keeping the definition of the language simple. The approach described here allows the definition of a domain specific language that is independent of a notation. The definition can be later specialized with an appropriate notation and, e.g., used for making UML models.

2.4 RBAC

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

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

A user u is granted the permission p if and only if $(u, p) \in \mathrm{AC}$. Aside from the technical question of how to integrate this relation into systems so that the granting of permissions respects this relation, a major challenge concerns how to effectively represent this information since just enumerating 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.,

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

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

$$\mathrm{AC} := \mathrm{PA} \circ \mathrm{UA}\,,$$

i.e.,

$$(\mathrm{u},\mathrm{p}) \in \mathrm{AC} \quad :\Longleftrightarrow \quad \begin{array}{l} \exists role \in \mathrm{Roles}: \\ (\mathrm{u},\, role) \in \mathrm{UA} \,\wedge (role, \mathrm{p}) \in \mathrm{PA}\,. \end{array}$$

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

$$\mathrm{AC} := \mathrm{PA} \circ \geq \circ\, \mathrm{UA}\;\;,$$

To express the same access control relation without a role hierarchy, one must, for example, put each user in additional roles, i.e., a user is then not just in his original roles, but also in all smaller roles. Alternatively, one can give roles additional permissions, i.e., a role not only has its assigned permissions, but also all permissions of 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.

Apart from addressing the problem of *scalability* mentioned above, RBAC also improves the *usability* of the access control configuration and its administration in that roles provide a convenient and intuitive abstraction that corresponds closely to the actual organizational structure of companies.

We have chosen RBAC as foundation of our security modeling language because it is well-established and supported by many existing technology platforms, which is a prerequisite for the model-driven and generative construction of secure systems. 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. 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 in later sections, our approach of generating architectures from models provides a means to overcome such limitations and differences in technologies.

2.5 Security Architectures

We use three different security architectures as examples of target platforms in this paper. We survey them here, focusing on their access control aspects.

2.5.1 Enterprise JavaBeans

Enterprise JavaBeans (EJBs) is a component architecture standard [19] for the development of 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 core of an EJB component (and the focal point of access control) is the *bean class*, which contains the business logic of the component. Clients access components using *home* or *remote interfaces*. Home interfaces are used for creating and finding bean instances whereas remote interfaces define the methods applicable to the component instances.

The access control model of EJB is based on RBAC, where the protected resources are the methods accessed using the interfaces of an EJB. An access control policy can mainly be realized using *declarative access control*. This means that 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 the definition of a permission 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. The listed roles are granted the right to execute the listed methods.

In general, for realistic applications the information needed to specify a comprehensive access control policy is quite voluminous and there is the danger that developers introduce errors due to oversights or shortcuts. For example, suppose a high-level security policy states that a role is granted the permission to read the state of a particular component. At the implementation level, this requires granting the role access to all read methods of the attributes and associations of the component, i.e., defining one method-permission element containing one method element for each of these read methods. To save time, a developer might just define one method permission granting the role full access to all methods of the EJB[1], which would be too liberal. As we indicate later, modeling security policies at a high abstraction level and automatically generating the corresponding deployment descriptors provides a promising solution to this problem.

In addition to declarative access control, EJB offers the possibility of implementing access control decisions within the business logic of components. This mechanism is called *programmatic access control* and is based on inserting appropriate 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.

2.5.2 Enterprise Services for .NET

The Microsoft Enterprise Services for .NET support the execution of server-side components based on the .NET platform by providing services such as distributed transactions, life-cycle management, and security.

The Enterprise Services support declarative and programmatic access control [5]. Here, programmatic access control allows one to obtain the identity of the caller of a method and to check its role assignments. The declarative mechanism supports the configuration of access control restrictions at the level of applications, components, interfaces, and methods. To achieve this, *.NET attributes* are added to the source code of a component, an interface,

[1]This can be achieved using the wild-card "*" as the method-name.

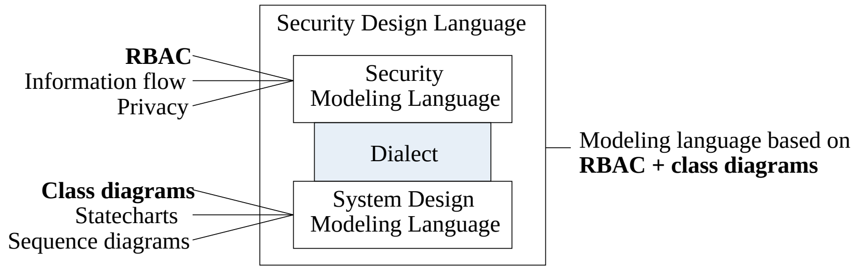

Figure 6: Security Design Language Schema

or to the *assembly descriptor* of an application. The following C# code fragment grants the role `Supervisor` the right to execute the method `cancel()`.

```
[SecurityRole("Supervisor")]
public void cancel(){...}
```

2.5.3 Java Servlets

The Java Servlet Specification [15] defines an execution environment for web components implemented in Java. These components are called *servlets*. A servlet is essentially 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.

A servlet environment, called the *servlet container*, supports 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

As explained in the introduction, we aim to close two gaps with model driven security: the gap between security models and system design models, and the gap between design and implementation. We accomplish this by a model-driven development process where security is explicitly integrated into the modeling language and supported during model transformation.

Rather than developing a single language for security modeling, we propose a schema for building such languages in a modular way. The overall form of our schema is depicted in Figure 6. 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 must define, as depicted in Figure 2, transformation functions mapping models (in the security design language) to security infrastructures. As we will indicate, it is possible to do this in a compositional way, where functions implementing transformations for model driven architecture on the underlying system design models are extended to generate security infrastructures from security policy specifications.

Below we discuss these aspects in more detail and, in subsequent sections, we consider particular language instances. Due to space requirements, we will focus on a security modeling language, which we call *SecureUML*, that is based on an extension of Role-Based Access Control. We will present this language in detail, emphasizing the general metamodeling ideas behind it, which can be used to define other security modeling languages. We will then present two different system design modeling languages and different bridging 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* (e.g., icons, strings, or geometric figures). To distinguish these two kinds of syntax, we call the underlying syntax the *abstract syntax* and the notation the *concrete syntax*. In general, the abstract syntax is formally defined, e.g., by grammars, whereas the notation is informally described. The translation between notation and abstract syntax is generally straightforward; we give examples in Section 4.2.

The definition of a language's syntax follows the standard approach taken in MDA, described in Section 2.3. The abstract syntax is defined by a metamodel, in accordance with the OMG Meta-Object Facility, and the concrete syntax is defined by a UML profile. This will be illustrated when defining SecureUML in Sections 4.1 and 4.2.

In our work, the semantics serves two purposes. First, it explains what specifications mean, e.g., when access is granted to protected resources. Second, it guides our translation and provides a basis against which we can judge its correctness. In Section 4.3, we define the semantics of SecureUML by defining a mapping from models into many-sorted first-order relational structures.

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

Our schema is open to different system design modeling languages. This supports the common practice of using domain-specific languages to specify systems using a vocabulary suitable for formalizing the system at different levels of abstraction and from different views. We give examples of such domain specific languages based on UML in Sections 5 and 8.

To make a design modeling language "security aware", we combine it with a security modeling language by merging their vocabularies at the level 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 based on using subtyping (in the object-oriented sense) to classify constructs in one language as belonging to subtypes in the other.

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. This depends 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;
- the rules for inheritance between resources;
- the default access control policy for actions where no explicit permission is defined (i.e., whether access is allowed or denied by default).

We give examples of integrating SecureUML into different system modeling languages in Sections 5.1 and 8.1.

3.3 Model Transformation

Given a language that is an instance of the schema in Figure 6, 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 particular system design modeling language is already equipped with a transformation function that can generate code, or system infrastructure from models (see the discussion of this in Section 2.3). Our task then is to define how the additional modeling constructs, from the security modeling language, are translated into system constructs. Our aim here isn't to generate or develop new kinds of security architectures, but rather to capitalize on the existing security mechanisms of a given component architecture and to automatically generate appropriate instances of these mechanisms. Of course, for this to be successful, the modeling constructs in the security modeling language should be designed with an eye open to the class of architectures and security mechanisms that will later be part of the target platforms.

We require that a transformation function is correct with respect to the semantics of the security modeling language, i.e., the transformation function must be *semantic preserving*. To judge the correctness thus also requires a formal semantics for the targeted security ar-

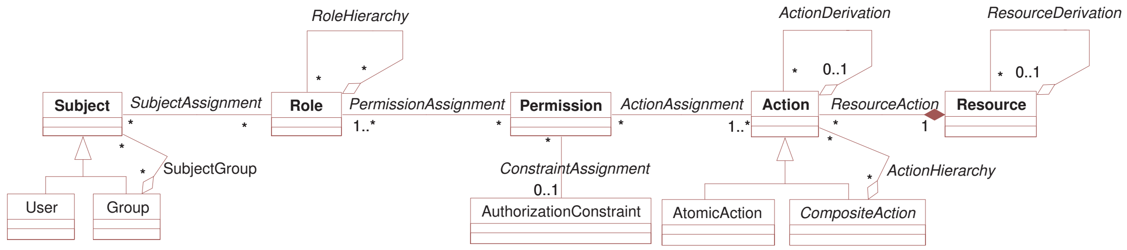

Figure 7: SecureUML Metamodel

chitectures.[2] In this work, we model the targeted security architectures at a high level of abstraction. Basically, in our model, system execution is abstracted to a sequence of attempts to perform protected actions. For every such attempt, a security monitor (1) checks the attempt against the static role and permission assignments (which are generated by the transformation function), and (2) evaluates code (again generated by the transformation function) that must return the Boolean `true` to allow the action. *Correctness* of the transformation function means that the security monitor allows an action if and only if the action is allowed according to the semantics of the security design language. If desired, this high-level notion of correctness could be refined and provide a basis for a fully verified mapping. However, this would involve, among other things, showing that the security monitor behaves as specified (e.g., according to the EJB specification) and that evaluation of code at runtime also behaves "as expected". We will not pursue this further, as it is outside the scope of this paper. Instead, we will sketch a proof, based on this simplified model, of the correctness of the transformation process using the example of the EJB platform (cf. Section 6.3).

We will illustrate the transformation process using SecureUML. We will define transformations that generate security infrastructures for platforms that support RBAC and programmatic access control. Specifically, we will give examples of transformation functions that translate models defined with the design modeling language *ComponentUML* (described in Section 5) into secure, executable systems for the component platforms EJB (Section 6) and .NET (Section 7) and a transformation function that translates models given in the design modeling language *ControllerUML* (defined in Section 8) into secure web applications based on the Java Servlet standard (Section 8.4).

4 SecureUML

We now define the abstract syntax, concrete syntax, and semantics of SecureUML.

4.1 Abstract Syntax

Figure 7 presents the metamodel that defines the abstract syntax of SecureUML. The language is based on RBAC, which we extend in several directions. Observe that the left-hand part of the diagram essentially formalizes RBAC, where we extend the *Users* (defined in Section 2.4) by *Groups* and formalize the assignment of users or 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*. These permissions may be constrained

[2]Unfortunately, most such architectures are not formally specified, so a rigorous correctness proof would also involve formalizing their behavior.

to hold only in certain system states by *authorization constraints*. Additionally, we introduce hierarchies not only on roles (which is standard for RBAC), but also on resources and 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. 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 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, e.g., dependent on the current time or attribute values. Consider a policy stating that an employee is allowed to withdraw money from a company account as long as 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, ordered in an ActionHierarchy, are used to group actions.

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 all of the (transitively) contained subactions. This semantics yields a simple basis for defining high-level actions. Suppose 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. Another reason for introducing action hierarchies is that they simplify the development of generation rules since it is sufficient to define these rules only for the atomic actions.

Inheritance plays a central role in object-oriented modeling languages. If two types are in an inheritance relationship then a subset of the features of the supertype, e.g. its methods, are *derived* along this relationship and are accessible within the context of the subtype. Types and features are resources in SecureUML and we must clarify how permissions on actions are derived along inheritance hierarchies.

The aggregations ResourceDerivation and ActionDerivation provide the foundation to support the derivation of permissions along inheritance hierarchies. The aggregation ResourceDerivation represents the derivation relation between resources. A SecureUML dialect defines which

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

Table 1: Mapping between SecureUML concrete and abstract syntax

subresources are derived when two resources are in an inheritance relationship. The aggregation ActionDerivation is determined based on the association ResourceDerivation. An action is derived from another action if their owning resources are in a derivation relation and both actions have the same name. As we will see in Section 4.3, the semantics of SecureUML states that a permission for an action also implies the permission for all derived actions.

Together the types Resource and Action formalize a generic resource model that serves as a foundation for integrating SecureUML into 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 expressions are given in a UML-based concrete syntax. For this purpose, we define a UML profile that formalizes the modeling notation of SecureUML using stereotypes and tagged values. In this section, we will 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: The notation for defining *resources* is left open and must be defined by the dialect; some information is automatically determined based on dialect rules (Action and its subtypes, ResourceAction, ActionHierarchy, ResourceDerivation, and ActionDerivation); and 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 an example model of roles and users, given in Figure 8, which formalizes the first part of the security policy introduced in Section 2.1. This example will allow us to explain some of the fine points of our concrete syntax in detail. It will also serve as the basis for defining access control policies for our running example.

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 by a UML generalization relationship. The role referenced by the arrowhead of the generalization relation is considered the superrole of the role referenced by the tail of the arrow, 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`.

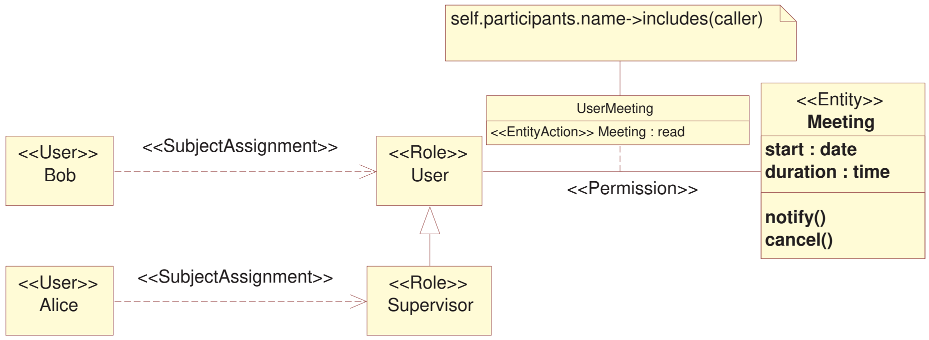

Figure 8: Example of the concrete syntax of SecureUML

Users are defined as UML classes with the stereotype «User» and groups are UML classes with the stereotype «Group». 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. The membership of a subject in a group is defined as a dependency with the stereotype «SubjectGroup». The group is referenced by 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`[3]

The right-hand part of Figure 8 specifies a permission on a protected resource. Specifying this is only possible after we have 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 statemachines 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 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 action *read*, i.e., we model that 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». We have chosen this representation as it is concise and it always satisfies the cardinality constraints on permissions, since an association class will be deleted when one of the referenced classes is removed from the model.

Such an association class connects a role (or several roles) 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 `UserMeeting` 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. "read". The resource name is stored in the tagged value identifier and references the

[3]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 in UML models, but rather using administration tools provided by the target platform.

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. In our example, the attribute of type "read" with the stereotype «EntityAction» and the identifier "Meeting" denotes the action *read* on the class `Meeting`. The stereotypes for action references and the naming conventions for identifiers are defined as part of the dialect.

Each authorization constraint is stored as an OCL expression in the tagged value constraint of the permission that it is constraining. 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 policy, the permission `UserMeeting` is constrained by the authorization constraint "`self.participants.name->includes(caller)`", which restricts the permission to users who are participants of a particular meeting.

4.3 Semantics

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

The basic elements of the access control configuration are the sets *Subjects*, *Users*, *Roles*, *Permissions*, and *Actions*, each containing entries for every model element of the corresponding metamodel types Subject, User, Role, Permission, and Action. Note that to satisfy the SecureUML metamodel, *Users* must be a subset of *Subjects.* We also have the relations $UA \subseteq \mathit{Subjects} \times \mathit{Roles}$, $PA \subseteq \mathit{Roles} \times \mathit{Permissions}$, and $AA \subseteq \mathit{Permissions} \times \mathit{Actions}$, which contain tuples for each instance of the corresponding association[4] in the abstract syntax of SecureUML.

Additionally, we define the partial orders $\geq_{\mathit{Subjects}}$, $\geq_{\mathit{Roles}}$, and $\geq_{\mathit{Actions}}$ on the sets of subjects, roles, and actions respectively. $\geq_{\mathit{Subjects}}$ is given by the reflexive, transitive closure of the aggregation association SubjectGroup in Figure 7 and formalizes that a group is larger than all its contained subjects. $\geq_{\mathit{Role}}$ 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. $\geq_{\mathit{Actions}}$ is defined by the reflexive, transitive closure of the union of the composition hierarchies on actions, defined by the aggregation ActionComposition, and the derivation hierarchy on action, defined by the aggregation ActionDerivation.

Given a system model (i.e., a UML model in a system design modeling language), we define a many-sorted signature $\Sigma = (S, F, P)$, where S is a set of sorts, F is a set of typed function symbols (including constants), and P is a set of typed predicate symbols. S contains one sort for each class in the system model and additionally the sort *String.* F contains a function symbol for each attribute or side-effect free method of the model that does not return a Boolean value, and the constant symbol self_C for each class C in the system model

[4] We generally abbreviate the names of the associations in Figure 7 as their capital letters and the association SubjectAssignment as UA.

denoting the object that is currently accessed. Additionally, there is a constant symbol *caller* of sort *String* denoting the name of the user on whose behalf an action is performed at a time point t. Finally, P contains a predicate symbol for every side-effect free method in the system model that returns a Boolean value.

Given an access control configuration, to define $\phi_{\mathrm{AC}}(u, a)$ we first define the auxiliary function UAP : $(\mathit{Users} \times \mathit{Actions}) \to 2^{\mathit{Permissions}}$ by

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

which determines the set of permissions a subject has for an action. In the absence of constraints, a user u has access to the action a if and only if u has a permission for a, i.e., if and only if $\mathrm{UAP}(u, a)$ is not empty. Expanding definitions, this means that access is allowed if and only if the user is assigned to a role that is larger or equal to a role that has a permission to an action that is superior or equal to the atomic action corresponding to executing this method. We also introduce the function Constraint : $\mathit{Permissions} \to \mathcal{L}^{\Sigma}$, which maps permissions to their associated constraints translated (the translation is given in the appendix) into the first-order language over Σ and associates the formulae *true* with unconstrained permissions. In this setting, we define that at time t the user u is allowed to perform the action a if and only if $\mathfrak{S}(t) \models \phi_{\mathrm{AC}}(u, a)$, where

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

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

A final point concerns default behavior. Existing technologies (cf. Section 2.5) differ in how they treat the case where no permission is explicitly given for an action. For example, in the Java Servlet architecture, access is granted by default, whereas access is denied in the Java platform security architecture [13]. We call this property of a platform its *default behavior*. To support different possibilities, while keeping the semantics simple, we add a distinguished element `DefaultPermission` to the set *Permissions* and a distinguished element `DefaultRole` to the set *Roles*. All users are automatically members of `DefaultRole`, and `DefaultPermission` is automatically assigned to all actions that do not have other permissions that are directly, or indirectly (via an *ActionComposition* or *ActionDerivation* relation), assigned. The default behavior, i.e., whether `DefaultRole` has `DefaultPermission`, is specified as part of a dialect and, as we will see, affects the model transformation process. If left unspecified in the dialect, we assume that access is granted by default.

5 Example Modeling Language: ComponentUML

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

ComponentUML is a simple modeling language that can be used to define distributed object-oriented systems. The metamodel for ComponentUML is shown in Figure 9. Elements of type Entity represent business object types of a particular domain. An entity may

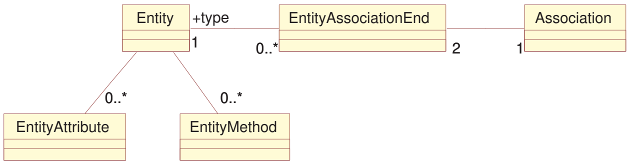

Figure 9: ComponentUML Metamodel

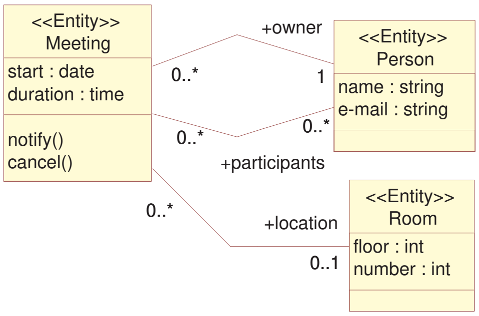

Figure 10: Scheduling application

have multiple methods and attributes, represented by elements of the types EntityMethod and EntityAttribute 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 EntityAssociationEnd.

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 as a respective method, attribute, or association end of the entity, so no further stereotypes are necessary.

Figure 10 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, represented by UML classes with the stereotype «Entity».

5.1 Extending the Abstract Syntax

Merging the syntax As the first step in 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 technically 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 while merging them.

Identifying protected resources Second, we identify the model elements of ComponentUML representing protected resources and formalize them 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. Moreover, when doing this, we must account for what can ultimately

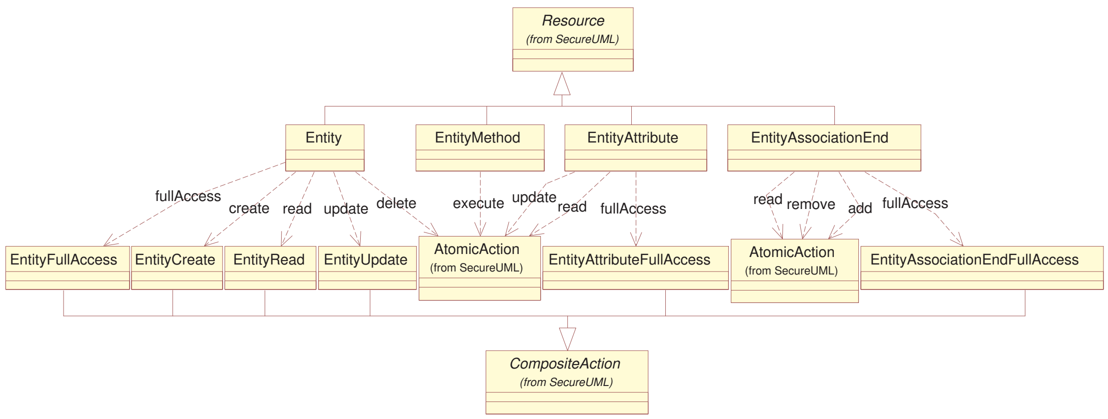

Figure 11: SecureUML Dialect for ComponentUML Metamodel

be protected by the target platform. Suppose, for example, we decide to interpret entity attributes as protected resources and the target platform supports access control on methods only. This is possible, but would necessitate 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, EntityMethod, EntityAttribute, and EntityAssociationEnd. This identification is technically realized by deriving these metatypes from the SecureUML type Resource as shown in Figure 11. In this way, the metatypes derive all properties needed to define authorization policies. Additionally, this figure shows several action classes that are defined as subtypes of the SecureUML class CompositeAction. This way, the action composition hierarchy can be defined as part of each action's type information by way of an OCL invariant (see below).

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 would result in an intuitive 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. However, we must ensure that these high-level actions can be mapped to actions in the target security platform. We usually define actions of both sorts and connect them using hierarchies.

In the metamodel, the set of actions a particular resource type offers is defined by named dependencies from the resource type to action classes, as shown in Figure 11. 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 11 formalizes that an EntityAttribute always possesses the action *fullAccess* of type EntityAttributeFullAccess and the actions *read* and *update* of type AtomicAction.

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, *execute* for all side-effect free methods of the entity.
EntityUpdate	*update* for all attributes of the entity, *add* and *delete* all association ends of the entity, *execute* for all none-side-effect free methods of the entity.
EntityAttributeFullAccess	*read* and *update* of the attribute.
EntityAssociationEndFullAccess	*read*, *add* and *delete* of the association end.

Table 2: SecureUML dialect action hierarchy

Defining the action hierarchy In 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 in 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. This example also shows the strong coupling between composite actions and a particular resource type as this expression depends on the structure of the resource type Entity.

Another example for the action class EntityRead is given by the OCL expression

```
context EntityRead inv:
 subordinatedActions =
  resource.attributes.actions->select(name="read")->union(
  resource.roles.actions->select(name="read"))->union(
  resource.operations->select(isQuery).actions->select(name="execute")).
```

This states that EntityRead is larger than the *read* actions of the attributes and association ends contained in the entity and the *execute* actions of all side-effect free methods of the entity. Our OCL formalization of this is somewhat complex as we must use the syntax of the metamodel to select actions of the resources that are contained in the entity instance that the action belongs to.

5.2 Extending the Concrete Syntax

In the previous section, we have seen how the abstract syntax of ComponentUML has been augmented with syntax for security modeling by combining it with the abstract syntax of SecureUML. Analogously, we must now extend the concrete syntax of ComponentUML. We achieve this 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

stereotype	resource type	naming convention
«EntityAction»	Entity	empty string
«EntityMethodAction»	EntityMethod	method signature
«EntityAttributeAction»	EntityAttribute	attribute name
«EntityAssociationEndAction»	EntityAssociationEnd	association end name

Table 3: Action Reference Types for ComponentUML

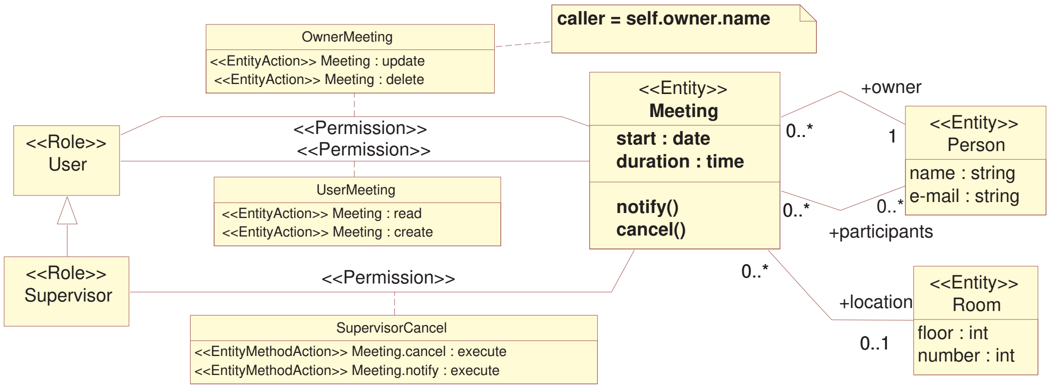

Figure 12: Scheduler Example with Authorization Policy

permission, which is any UML class in the SecureUML notation (see Section 4.2), is restricted to UML classes with the stereotype «Entity». Finally, we define the action reference types for entities, attributes, methods, and association ends as shown in Table 3.

5.3 Modeling the Authorization Policy

We now use the combined language to formalize the security policy stated in Section 2.1. We do this by defining permissions in the scheduler entity model, which correspond to the three policy requirements. As these permissions associate roles with actions, we employ the extension with the roles `User` and `Supervisor`, 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 12, which grants the role `User` the right to perform *create* and *read* 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* on `Meeting`. Additionally, we restrict this permission with the authorization constraint "caller = self.owner.name", which states that the name of a caller must be equal to the name of the owner of the particular meeting instance. Due to the definition of the action `update` (cf. Table 2), this permission must hold for all attempts to change the value of the attributes or association ends of the meeting entity as well as for invocations on the methods `notify` or `cancel`.

To complete the formalization of our security policy, 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`.

5.4 Semantics

We now illustrate the semantics by analyzing several access control decisions for the users Alice and Bob, each trying to execute the method `cancel` on `Meeting`. Here we assume that our dialect has the *default behavior* "access allowed" and we directly apply the semantics of SecureUML to the example policy given in the previous section. The (partial) example access control configuration is

$$\begin{aligned}
\text{Users} &:= \{\text{Bob}, \text{Alice}, \text{Jack}\} \\
\text{Roles} &:= \{\texttt{User}, \texttt{Supervisor}\} \\
\text{Permissions} &:= \{\texttt{OwnerMeeting}, \texttt{SupervisorCancel}, \dots\} \\
\text{Actions}^{5} &:= \{\texttt{Meeting.update}, \texttt{Meeting.cancel.execute}, \dots\} \\
\text{SA} &:= \{(\text{Bob}, \texttt{User}), (\text{Alice}, \texttt{Supervisor})\} \\
\text{PA} &:= \{(\texttt{User}, \texttt{OwnerMeeting}), (\texttt{Supervisor}, \texttt{SupervisorCancel}), \dots\} \\
\text{AA} &:= \{(\texttt{OwnerMeeting}, \texttt{Meeting.update}), \\
&\quad\ (\texttt{SupervisorCancel}, \texttt{Meeting.cancel.execute}), \dots\} \\
\geq_{Roles} &:= \{(\texttt{Supervisor}, \texttt{User}), (\texttt{Supervisor}, \texttt{Supervisor}), (\texttt{User}, \texttt{User})\} \\
\geq_{Actions} &:= \{(\texttt{Meeting.update}, \texttt{Meeting.cancel.execute}), \dots\}\ ,
\end{aligned}$$

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

$$\begin{aligned}
S &:= \{\mathit{Meetings}, \mathit{Persons}, \dots\} \cup \{\texttt{String}\} \\
F &:= \{\mathit{self}_{\mathit{Meeting}}, \dots, \mathit{MeetingOwner}, \mathit{PersonName}\} \cup \{\texttt{caller}\} \\
P &:= \{\}\ .
\end{aligned}$$

The constant symbol $\mathit{self}_{\mathit{Meeting}}$ of sort *Meeting* denotes the currently accessed meeting. The function symbols $\mathit{MeetingOwner}\colon \mathit{Meetings} \to \mathit{Persons}$ and $\mathit{PersonName} : \mathit{Persons} \to \mathit{String}$ 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 at time t_1. The corresponding structure $\mathfrak{S}(t_1)$ over the signature Σ is given by the interpretation where

$$\begin{aligned}
\texttt{caller}^{\mathfrak{S}(t_1)} &:= \text{Alice} \\
\mathit{Meetings}^{\mathfrak{S}(t_1)} &:= \{\mathit{meeting}_{\text{Jack}}\} \\
\mathit{Persons}^{\mathfrak{S}(t_1)} &:= \{\mathit{alice}, \mathit{bob}, \mathit{jack}\} \\
\mathit{self}^{\mathfrak{S}(t_1)}_{\mathit{Meetings}} &:= \mathit{meeting}_{\text{Jack}} \\
\mathit{MeetingOwner}^{\mathfrak{S}(t_1)} &:= \{(\mathit{meeting}_{\text{Jack}}, \mathit{jack})\} \\
\mathit{PersonName}^{\mathfrak{S}(t_1)} &:= \{(\mathit{jack}, \text{Jack}), (\mathit{bob}, \text{Bob}), (\mathit{alice}, \text{Alice})\}\ .
\end{aligned}$$

The formula that has to be satisfied by the structure $\mathfrak{S}(t_1)$ in order to grant Alice access is built according to the definition (2), given in Section 4.3. The set of permissions Alice has on the action `Meeting.cancel.execute` is

$$\text{UAP}(\text{Alice}, \texttt{Meeting.cancel.execute}) = \{\texttt{OwnerMeeting}, \texttt{SupervisorCancel}\}\ .$$

The constraint expression `self.owner.name = caller` on the permission `OwnerMeeting` is translated into the formula $\texttt{caller} = \mathit{PersonName}(\mathit{MeetingOwner}(\mathit{self}_{\mathit{Meetings}}()))$ in the

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

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

$$\mathfrak{S}(t_1) \models true \vee \texttt{caller} = PersonName(MeetingOwner(self_{Meetings}())) ,$$

which is satisfied.

Alternatively, suppose that Bob tries to perform this action at time t_2. The corresponding structure $\mathfrak{S}(t_2)$ differs from $\mathfrak{S}(t_1)$ by the interpretation of the constant symbol `caller`, which now refers to "Bob". Bob only has the permission `OwnerMeeting` for this action, i.e.

$$\mathrm{UAP}(\mathrm{Bob}, \texttt{Meeting.cancel.execute}) = \{\texttt{OwnerMeeting}\} .$$

Hence

$$\mathfrak{S}(t_1) \models \texttt{caller} = PersonName(MeetingOwner(self_{Meetings}()))$$

is required for access. Since Jack 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 and illustrate the approach using the example introduced in the previous section. 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 or 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>
```

An EntityMethod is transformed to a method declaration in the component interface of the respective entity bean and a method stub in the corresponding bean implementation class. The following shows the stub for the method `cancel` of the entity `Meeting`.

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

For each EntityAttribute, access methods for reading and writing the attribute value are generated along with a "persistency configuration" 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);
```

The transformation applied to elements of type EntityAssociationEnd is similar to the rules defined for attributes. Three access methods are generated for reading the value, adding new references to the association end, and removing existing references. Furthermore, the persistency configuration for storing the association-end data in a database is constructed. The following code fragment shows the declarations of the access methods for the association end `participants` of the entity `Meeting`.

```
Collection getParticipants();
void addToParticipants(Person participant);
void removeFromParticipants(Person participant);
```

6.2 Generating Access Control Infrastructures

From a security design model, an EJB security infrastructure is generated 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 assigned 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 do not need to consider actions without permissions in the generation process.

6.2.1 Generating Permissions

As explained in Section 2.5.1, a method permission element names the set of roles and the set of EJB methods that these roles are allowed to access. Generating a method permission can therefore be split into two parts: generating a set of roles and assigning methods to them.

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

rule #	resource type	action	EJB methods
1	Entity	*create*	automatically generated factory method
2	Entity	*delete*	delete methods in component and home interface
3	EntityMethod	*execute*	corresponding method
4	EntityAttribute	*read*	get-method of the attribute
5	EntityAttribute	*update*	set-method of the attribute
6	EntityAssociationEnd	*read*	get-method of the association end
7	EntityAssociationEnd	*add*	add-method of the association end
8	EntityAssociationEnd	*delete*	remove-method of the association end

Table 4: Atomic action to method mapping for EJB

subactions (under the transitive closure of the relation defined by the union of the associations ActionComposition and ActionDerivation) 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 atomic actions 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

```
Actions := {(Meeting, create), (Meeting, read)} .
```

The resulting set of atomic actions is

```
Actions := {(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 given in Figure 13.

6.2.2 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.

We proceed as follows. First, given a method m, the atomic action a corresponding to the method is determined using the 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 transitive closure of the order relation defined by the union of the associations ActionComposition and ActionDerivation). Next, for each permission p in the resulting set ActionPermissions(a), the set PR(p) of roles assigned to this permission 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 of the following form is generated

```
<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>
```

Figure 13: Generated XML code for the methods of the permission **UserMeeting**

$$\begin{array}{l}\texttt{if (!(} \bigvee_{p \in \mathrm{ActionPermissions(a)}} \Big(\Big(\bigvee_{r \in \mathrm{PR}(p)} \mathrm{UserRole(r)} \Big) \wedge \mathrm{Constraint}(p) \Big) \texttt{))} \\ \qquad \texttt{throw new AccessControlException("Access denied."); .}\end{array} \tag{3}$$

This scheme is similar to Equation (2) on page 18, which defines $\phi_{\mathrm{AC}}(u, a)$ in Section 4.3, as each permission represents an (alternative) authorization to execute an action. However, because the user is not known at compile-time, we must consider here all permissions for all users on this action, which are determined by the set ActionPermissions(a). Also, we must consider for each permission p whether the user is assigned to one of the roles assigned to this permission (denoted by PR(p)). If a constraint is assigned to this permission, it is evaluated afterwards. Access denial is signaled to the caller by throwing an exception.

As an example, the following shows the assertion generated for the method **Meeting::cancel**.

```
if (!(ctxt.isCallerInRole("Supervisor") /* SupervisorCancel */
      ||
     ((ctxt.isCallerInRole("User") || ctxt.isCallerInRole("Supervisor"))
     && ctxt.getCallerPrincipal.getName().equals(getOwner()))
    )) 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>)**. **ctxt** references an object of type javax.ejb.EJBContext, which 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**, which represents the name of the current 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.

6.3 Correctness of Generation

As stated in Section 3, judging the correctness of the transformation process requires a formal semantics for the targeted security architecture. In the following we first give an informal semantics of the security architecture of EJB, which can be further formalized. Afterwards, we explain why systems that are generated according to the rules given above actually implement the access control policy that is defined by the formal semantics of SecureUML.

6.3.1 Informal Semantics of EJB Security Architecture

In the EJB context, the set of protected resources is given by the set of the methods of the entity beans. Each method provides the single action to "execute this method", the union of which form the set of actions. Permission to perform these actions can be denied in two cases. First, if the execution of a method is restricted by at least one method permission element in the deployment descriptor, a user may only execute this method if he has one of the roles listed in one of the method permission elements protecting the method. Note that EJB does not support role hierarchies here. Second, in the body of methods there can be additional code whose evaluation decides if execution of this method should be allowed, i.e., there can be an assertion of the form

```
if( <predicate> ){
  throw new AccessControlException("Access denied.");
} ,
```

where `<predicate>` is defined by the application developer. In all remaining cases, method execution is allowed.

6.3.2 Correctness Proof

To argue the correctness of the generation rules with regard to an arbitrary atomic action a, we distinguish three cases:

1. There is no permission assigned directly to a, or to an action a' with $a' \geq_{Actions} a$, in the security design model.
2. There is at least one permission assigned, either directly or indirectly, to a in the security design model, but none of these permissions is assigned an authorization constraint.
3. There is at least one permission assigned, either directly or indirectly, to a in the security design model, and at least one of these permissions has an authorization constraint assigned.

In the first case, the generation rules generate neither a method permission nor an assertion. The default behavior of the SecureUML dialect for ComponentUML is "access allowed" and the EJB container allows execution of the relevant methods corresponding to this atomic action, which is correct.

In the second case, the formal semantics of SecureUML specifies that

$$\phi_{\mathrm{AC}}(u,a) = \begin{cases} \mathit{true} & \text{if } \mathrm{UAP}(u,a) \neq \emptyset \\ \mathit{false} & \text{if } \mathrm{UAP}(u,a) = \emptyset \end{cases} ,$$

i.e., access is allowed if and only if the user is assigned to a role that is larger than or equal to a role that has a permission to an action that is larger than or equal to the atomic action corresponding to executing this method (cf. Equations (1) and (2) on page 18 in Section 4.3). As no authorization constraint is assigned, an assertion is not generated. Therefore we only need to show that a user is in a role listed in the generated method-permission elements if and only if $\mathrm{UAP}(u, a)$ is not empty. However in the generation of these method-permission elements, both the hierarchy on roles as well as the hierarchy on actions are expanded when calculating the set of roles and the set of methods that appear in the method-permission element. This means that the method-permission that is generated for a permission p and contains the method corresponding to the action a, contains a role that the user u has if and only if $p \in \mathrm{UAP}(u, a)$.

In the third case, it suffices to show that the predicate `<predicate>` in the generated assertion evaluates to *true* if and only if $\phi_{\mathrm{AC}}(u, a)$ evaluates to *false*. This is equivalent to showing that

$$\bigvee_{p \in \mathrm{ActionPermissions(a)}} \Big(\Big(\bigvee_{r \in \mathrm{PR}(p)} \mathrm{UserRole}(r) \Big) \wedge \mathrm{Constraint}(p) \Big) \tag{4}$$

evaluates to *true* if and only if

$$\bigvee_{p \in \mathrm{UAP}(u,a)} \mathrm{Constraint}(p) \tag{5}$$

evaluates to *true*. However, comparing the definitions of $\mathrm{ActionPermissions}(a)$ and $\mathrm{PR}(p)$ with the definition of $\mathrm{UAP}(u, a)$, one sees that both formulae are essentially the disjunction over the same set of constraints $\mathrm{Constraint}(p)$. More precisely, there appear more constraints in Formula 4 than in Formula 5. These additional constraints are connected to permissions that the current user does not have. However, these additional constraints are nullified by the role checks $\mathrm{UserRole}(r)$, which are *false* for permissions the user does not have.

7 Generating a .NET System

One of the advantages of model driven security is that by implementing different translation functions one can generate security architectures for different platforms. Here we consider generating secure applications based on the programming language C# and the Enterprise Services for .NET, described in Section 2.5.2. Rather than presenting this translation in detail, we focus on the main conceptual differences to the EJB translation.

An Entity of ComponentUML is transformed into a *serviced component* of the enterprise services. The generated component consist of an interface and an implementation class; a default constructor is generated as well. This is shown by the following code fragment for the entity `Meeting`.

```
public interface IMeetingInterface{...}
public class Meeting : ServicedComponent, IMeetingInterface
{
  public Meeting(){...}
  ...
}
```

Methods and association ends are transformed to access methods and members as described in Section 6. Attributes are handled differently; for each attribute a *C# property* is added to the interface and the implementation class. The declaration of the property for the attribute `duration` of the entity `Meeting` is shown by the following example.

```
int duration
{
  get;
  set;
}
```

In contrast to the EJB generation, the transformation of permissions is "method-centric" because access restrictions are defined in .NET using `SecurityRole` attributes in the component source code (see Section 2.5.2). Such an attribute must be generated for each role that is allowed to execute a method m. The set of roles $MethodRoles(m)$ that are granted access to m is determined as follows. First, the action a corresponding to the method m is determined and the set of permissions $ActionPermissions(a)$ is calculated according to the rules given in Section 6.2.2. Second, for each permission in $ActionPermissions(a)$, all roles referenced by the association PermissionAssignment and all of their subroles (under the transitive closure of the relation defined by association RoleHierarchy) are added to $MethodRoles(m)$. For each role in $MethodRoles(m)$, a corresponding .NET attribute of type `SecurityRole` is generated as shown below for the method `Meeting::cancel`:

```
[SecurityRole("User")]
[SecurityRole("Supervisor")]
public void cancel() {...}
```

Note that there is no transformation rule for SecureUML roles because .NET does not require global role definitions. Instead, the .NET environment determines this information by analyzing the declared role checks of all components of a particular application.

The transformation of authorization constraints is analogous to the EJB transformation. There are only syntactical differences in the mapping rules between OCL and the C# programming language and the programmatic access control functions of .NET. The following shows the counterpart of the example given in Section 6.2.2.

```
if (!(ctxt.IsCallerInRole("Supervisor") /* SupervisorCancel */
      ||
     (ctxt.IsCallerInRole("User") || ctxt.IsCallerInRole("Supervisor"))
     && ctxt.OriginalCaller.AccountName == owner)
   )) throw new UnauthorizedAccessException("Access denied.");
```

8 ControllerUML

To demonstrate the general applicability of our approach, we now present a second design modeling language. This language, which we call *ControllerUML*, is based on statemachines.[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 [12]. 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 statemachines and the modeling language ControllerUML utilizes UML statemachines for that purpose.

The abstract syntax of ControllerUML is defined by the metamodel shown in Figure 14. 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

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

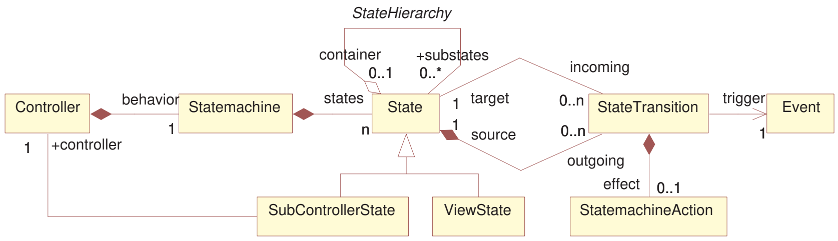

Figure 14: Metamodel of ControllerUML

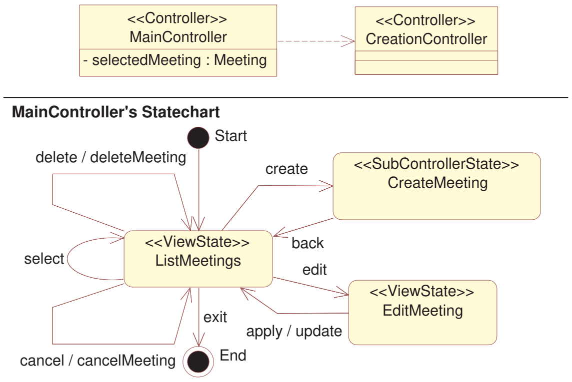

Figure 15: Controllers for Scheduling Application

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 statemachine action is a specification of an executable statement that is performed on entities of the application model. ViewState and SubControllerState are subclasses of State. A ViewState represents a state where the application interacts with a human by way of view elements like dialogs or forms. The view elements generate events in response to user actions, e.g. clicking a mouse button, that are processed by the controller's statemachine. A SubControllerState references another controller using the association end controller. The referenced controller takes over the control flow of the application when the referencing SubControllerState is activated. This supports the modular specification of controllers.

The notation of ControllerUML uses primitives from UML class diagrams and statecharts. An example model is shown in Figure 15. A Controller is represented by a UML class with the stereotype «Controller». The statemachine of the controller is defined by a UML statemachine that is attached to this UML class. States, transitions, events, and actions are represented by their natural 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 event names to identify transitions in our explanations. View states and subcontroller states are labeled with the respective stereotypes «ViewState» and «SubControllerState».

Figure 15 shows the design model for an interactive application that formalizes the scheduler workflow presented in Section 2.2. The controller class `MainController` is the

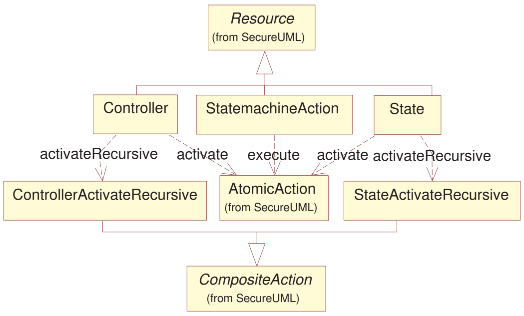

Figure 16: Resource Model of ControllerUML

top-level controller of the application and `CreationController` controls the creation of new meetings (details are omitted here to save space). The statemachine of `MainController` is similar to that of Figure 4. In the state `ListMeetings`, a form is displayed that shows all meeting entries in the database, independent of their owner. A user can trigger different actions from this form. It is possible to select a meeting and to execute an action on it. The selected meeting is stored in the attribute `selectedMeeting` of the controller object. An event of type `delete` triggers the execution of the action `deleteMeeting`, whereas `cancel` causes the execution of the action `cancelMeeting`. The transition `edit` causes a state transition to `EditMeeting`, where the user can change the meeting information. The action `update` on the outgoing transition `apply` of this state propagates the changes to the database. The creation of a new meeting is triggered by an event of type `create`. In this case, the subcontroller state `CreateMeeting` is activated, which in turn activates a controller of type `CreationController`. The reference from the subcontroller state `CreateMeeting` to the controller `CreationController` is not visible in the diagram. Instead, this information is stored in a tagged value of the subcontroller state.

8.1 Extending the Abstract Syntax

There are various ways to introduce access control into a process-oriented modeling language like ControllerUML. Each way 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 14) 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 16 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 statemachine actions of the outgoing transitions of the state, and the actions *activateRecursive* on all substates of the state. The respective OCL definition is as follows:

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

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

```
name = "execute")->union(
resource.substates.actions->select(name = "activateRecursive")))
```

This expression is built using the vocabulary defined by the ControllerUML metamodel as shown in Figure 14 and the dialect definition given in Figure 16. The third line accesses the resource the action belongs to (always a state) and selects the action with the name "activate". The next line first queries all outgoing transitions on the state and selects those transitions with an assigned statemachine action (association end effect). Afterwards, for each statemachine action, its (SecureUML) actions with the name "execute" is selected. The last line selects all actions with name "activateRecursive" on all substates of the state the action of type StateActivateRecursive belongs to.

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 statemachine.

8.2 Extending the Notation

First, we combine the notation of both languages by combining the SecureUML notation with that of ControllerUML. Afterwards, well-formedness rules are defined on SecureUML primitives to restrict their possible relations to ControllerUML elements representing protected resources. For example, the scope of a permission is restricted to UML classes with the stereotype «Controller». Finally, we define the action reference types for controllers, states, and statemachine actions, as shown in Table 5.

8.3 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 17 shows, we use two permissions to formalize the first requirement that all users should be 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 the right to activate the state `EditMeeting`. This permission is restricted by the constraint "`self.selectedMeeting.owner.name = caller`".

In the example policy, only supervisors are allowed to cancel any meeting. Therefore, the permission `SupervisorCancel` grants the role the unrestricted right to execute the action

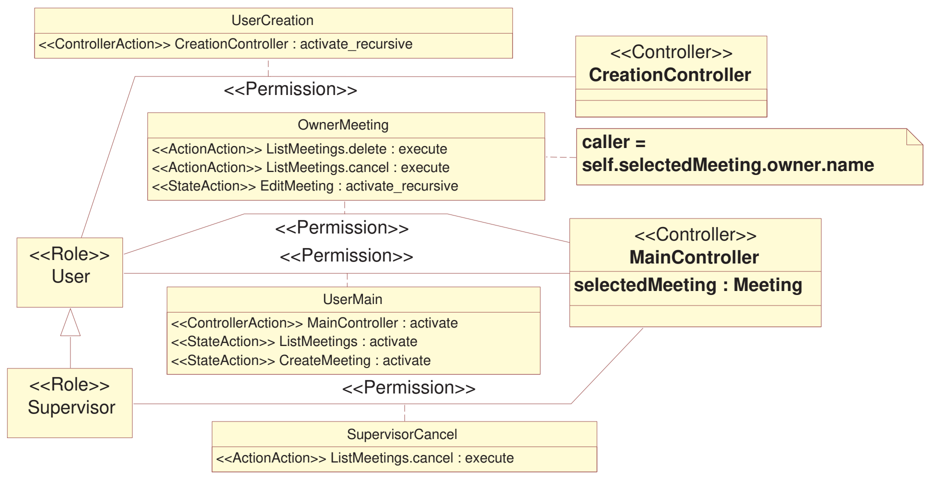

Figure 17: Policy for Scheduling Application

`cancelMeeting` on the transition `cancel`.

8.4 Transformation to Web Applications

In this section, we describe a transformation function that constructs secure web applications from ControllerUML models. As a basis, we assume the existence of a transformation function that converts UML classes and statemachines into controller classes for web applications, executed in a Java Servlet environment (see Section 2.5.3). We describe here how we extend such a function to generate a security infrastructure from SecureUML model elements.

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 modern web applications that are built from multiple servlets, with one acting as the central entry point to the application. This entry point servlet acts as a dispatcher in that it receives all requests and forwards them (depending on the application state) to the other servlets, which execute the business logic. The standard authorization mechanism only provides protection for the dispatcher. To overcome this weakness, we generate access control infrastructures that exploit the programmatic access control mechanism 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, which converts UML classes and statemachines 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. In the first step, the set ActionPermissions(a), which contains all permissions affecting the execution of an action, is determined in the way described in Section 6.2.2. Afterwards, an assertion is generated of the form:

$$\texttt{if (!(} \bigvee_{p \in \text{ActionPermissions}(a)} \Big(\Big(\bigvee_{r \in PR(p)} \text{UserRole}(r) \Big) \wedge \text{constraint}(p) \Big) \texttt{))} \qquad \texttt{c.forward("/unauthorized.jsp");} \tag{6}$$

The rule that generates this assertion has a structure similar to rule 3 in Section 6.2.2, 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 assertion generated for the execution of the action `cancel` on state `ListMeetings` is

```
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(Role)* on the request object and each constraint is translated into a Java expression, which accesses the members and side-effect free methods of the controller. The symbol *caller* is translated into a call to *getRemoteUser()* on the request object.

9 Related Work

Various extensions to the core RBAC model have been presented in the literature. The need for flexible constraints on role assignments to express different kinds of high-level organizational policies, like separation of duty, is emphasized by [16]. A formal language to express these constraints, based on first-order logic, is, for example, proposed by [6]. Ahn and Sandhu develop the "RSL99 language for Role-Based Separation of Duty Constraints" [1] and the Role-Based Constraint Language RCL2000 [2]. [4] show how these constraints can be expressed using OCL. In contrast to these works, we use authorization constraints as additional restrictions on the permissions that a role has. As a result, SecureUML can (unlike RBAC) be used to express access control policies that depend on the system state.

[3] give a description of the static, functional, and dynamic view of RBAC using UML diagrams. In contrast, our SecureUML metamodel provides a static view of our RBAC extensions. However, we can combine SecureUML with other design modeling languages and use the results to develop systems with access control infrastructures using security design models that support the formalization of different system views.

In the area of using UML for modeling security and access control, [9] show how UML can be used to model RBAC-like situations, in particular the RBAC Framework for Network Enterprises (FNE). Although the authors also use a UML-based notation to express access control policies, their syntax is different from SecureUML. Furthermore, we propose an approach for integrating policy models into system design models and facilitate this by allowing the definition of authorization constraints on the system state. Also, they don't consider the question of implementing infrastructures for enforcing access control policies, where we propose a generative approach.

Jürjens [17, 18] proposes 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 from them or for analyzing these policies.

Probably the most closely related work is the Ponder Specification Language [8, 7], 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 a RBAC-like way where one may specify roles and define role-based policy rules. Ponder also allows rules to be restricted by conditions expressed in a subset of OCL. Policies given in the Ponder Specification Language can directly be interpreted and enforced by a policy management platform. As an alternative, the authors propose using code generators to create infrastructures for particular access control technologies.

There are, however, important differences. To begin with, 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 (e.g., updating an object's state) can be defined using action hierarchies. Second, while Ponder is given an operational semantics, we employ a denotational semantics directly based on our RBAC extensions. Finally, and most importantly, 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.

10 Conclusion and Future Work

We have proposed model driven security as a methodology for developing secure systems. In doing so, we have developed a number of new ideas including: the use of object-oriented metamodels and dialects to formalize and combine modeling languages; a substantial extension of RBAC that serves as the semantics for our security modeling language SecureUML; and techniques for generating platform-specific security infrastructures. We have given examples of language combinations that illustrate our methodology as well as its application.

There are a number of promising directions for future work. To begin with, the languages we have presented constitute three different, representative examples of security and design modeling languages. There are many interesting 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, e.g., other UML diagram types (like Use Case Diagrams or Sequence Diagrams), which would support modeling different views of systems at different levels of abstractions. 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 have given SecureUML a first-order semantics based on relational and state-oriented extensions of RBAC. Our semantics has the advantage of being fairly simple: everything is formalized using basic concepts from many-sorted, first-order logic. Formalizing access control this way, by focusing on particular time points, is adequate to explain the operational decisions made during system execution. However, it has the disadvantage that one cannot reason about and compare the behavior of a system at different time points. To support this, one should probably move to a richer formalism like a first-order temporal logic.

We have used our semantics both to clarify what models mean and to reason about the correctness of code generation. However, we have just scratched the surface of what is possible here and we believe that model driven security can play a central role in analyzing and certifying secure systems. Since our models are formal, we can ask questions about them and get well-defined answers, like the examples given in Section 5.4. More complex kinds of analysis should be possible too. Just as some CAD tools have built-in model checkers for analyzing properties of designs, it should be possible to provide mechanical support, within a CASE tool, for calculating with security design models or analyzing safety and other kinds of invariance properties. For example, we might calculate a symbolic description of those states that allow Alice to withdraw money from her account, or, alternatively, prove that in all system states Alice cannot withdraw money from Bob's account. Moreover, the semantics can also be used to provide a basis for formally verifying the correctness of the code generators.

Finally, the question remains of how model driven security can be integrated into the overall system development process. For example, how roles and protected resources are identified during requirements analysis and incorporated into different models and how security requirements are refined during the different analysis and design phases. More experience carrying out large case studies should help shed light on the answer here.

APPENDIX

Here we sketch details concerning the semantics of authorization constraints, which were omitted in the body of the paper. In Section A.1 we clarify how to interpret the state of the system as a first-order structure, and in Section A.2 we provide details on the translation of OCL expressions to first-order syntax.

A.1 System state as a first-order structure

Given a system state at time t, the definition of a first-order structure $\mathfrak{S}(t)$ over the signature Σ (cf. Section 4.3) is straightforward and includes:

- the sorts D_i, $0 \leq i < \#datatypes$, given by the datatypes of the system, where each sort contains all values of the corresponding datatype;
- functions f_i^j, $0 \leq j < \#datatypes$, $0 \leq i < \#functions(D_j)$, given by all the side-effect free methods that do not return a `Boolean`;
- the relations P_i, given by all side-effect free methods that return a `Boolean`.

The interpretation of the symbols in Σ (cf. Section 4.3) over this structure is also straightforward when one considers the close correspondence between the symbols and their counterparts in the structure $\mathfrak{S}(t)$. The only non-trivial interpretations are those of the constant symbols *caller* and $self_i$. The interpretation of *caller* is given by the name of the subject that represents the current caller, i.e., the subject that is currently trying to access some protected resource. Let D_j be the sort representing the type of the object that is currently accessed. $self_j$ is interpreted by the element of the set D_j that represents the object that is currently accessed, and $self_i$ is interpreted arbitrarily for all remaining $i \neq j$.

Note that we give a sensible interpretation only for the single constant symbol $self_j$ that is of the same sort as the object that is currently accessed. However, this does not lead to problems because the translation from OCL constraints into this language (described below) ensures that no formulae arise containing symbols whose interpretation is arbitrary.

A.2 OCL translation

Our semantics for authorization constraints is completed by giving a function

$$[\![\cdot]\!] : \mathcal{L}_{\mathrm{OCL}} \to \mathcal{L}_{\mathrm{FOL}} \ ,$$

translating OCL syntax to first-order syntax. The syntax of OCL is unfortunately complex and a full semantics is out of the scope of the current paper. Hence we limit ourselves to a representative subset of OCL and sketch the main ideas.

The authorization constraints we consider here are OCL expressions of type `Boolean` and our translation is defined by recursion on the syntax of formulae. Boolean expressions are recursively built by using the infix operators **`and`**, **`or`**, **`xor`**, **`not`**, and **`implies`**, or they are built from other OCL expressions by using equality (`=`), or the comparison operators `<>`,`<`, `<=`, `>`, and `>=` (which can be user-defined for user-defined types).

Boolean expressions are translated directly to their first-order counterparts, e.g.,

$$\begin{aligned} [\![\phi_1 \ \texttt{and} \ \phi_2]\!] &:= [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\ [\![\phi_1 \ \texttt{xor} \ \phi_2]\!] &:= [\![\phi_1]\!] \vee [\![\phi_2]\!] \wedge \neg([\![\phi_1]\!] \wedge [\![\phi_2]\!]) \ . \end{aligned}$$

Equality of Boolean expressions is translated as

$$[\![t_1 \ \texttt{=} \ t_1]\!] := [\![t_1]\!] \leftrightarrow [\![t_2]\!] \ ,$$

and equality of expression of other types is simply translated as

$$[\![t_1 \ \texttt{=} \ t_1]\!] := [\![t_1]\!] = [\![t_2]\!] \ .$$

The translation of the comparison operators uses the corresponding first-order relations, e.g.,

$$\begin{aligned} [\![t_1 \ \texttt{<>} \ t_2]\!] &:= \ \neq_s ([\![t_1]\!], [\![t_2]\!]) \\ [\![t_1 \ \texttt{<} \ t_2]\!] &:= \ <_s ([\![t_1]\!], [\![t_2]\!]) \ , \end{aligned}$$

where s is the sort corresponding to the type of the expressions t_1 and t_2.

The base cases for the translation of OCL expressions of general type are the translations of `self` and `caller`:

$$\begin{aligned} [\![\texttt{self}]\!] &:= \mathit{self}_s \\ [\![\texttt{caller}]\!] &:= \mathit{caller} \ , \end{aligned}$$

where s is the sort corresponding to the type of the context of the expression.

The recursive case is given by

$$[\![\mathrm{term.function}(\mathrm{arg}_1, \ldots, \mathrm{arg}_n)]\!] := f([\![\mathrm{term}]\!], [\![\mathrm{arg}_1]\!], \ldots, [\![\mathrm{arg}_n]\!]) \ ,$$

where f is the function symbol (or predicate symbol, in case of a function returning a `Boolean`) corresponding to the function "function" belonging to the term "term".

References

[1] Gail-Joon Ahn and Ravi S. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 43–54. ACM Press, 1999.

[2] Gail-Joon Ahn and Ravi S. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, November 2000.

[3] Gail-Joon Ahn and Michael Eonsuk Shin. UML-based representation of role-based access control. In *9th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 195–200. IEEE Computer Society, June 2000.

[4] Gail-Joon Ahn and Michael Eonsuk Shin. 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, June 2001.

[5] Derek Beyer. *C# COM+ Programming*. John Wiley & Sons, book and cd-rom (october 15, 2001) edition, 2001.

[6] Fang Chen and Ravi S. Sandhu. Constraints for role-based access control. In *Proceedings of the first ACM Workshop on Role-based access control*, pages 39–46. ACM Press, 1996.

[7] Nicodemos Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College, University of London, 2002.

[8] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In Morris Sloman, Jorge Lobo, and Emil C. Lupu, editors, *Policies for Distributed Systems and Networks (POLICY 2001)*, number 1995 in LNCS, pages 18–38. Springer-Verlag, January 2001.

[9] Pete Epstein and Ravi S. Sandhu. Towards a UML based approach to role engineering. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 135–143. ACM Press, 1999.

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

[11] David S. Frankel. *Model Driven Architecture™ : Applying MDA™ to Enterprise Computing*. John Wiley & Sons, 2003.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Orientated Software*. Addison-Wesley, 1995.

[13] Li Gong. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1st edition, 1999.

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

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

[16] Trent Jaeger. On the increasing importance of constraints. In *Proceedings of fourth ACM workshop on Role-based access control*, pages 33–42. ACM Press, 1999.

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

[18] Jan Jürjens. UMLsec: Extending UML for secure systems development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer-Verlag, 2002.

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

[20] Object Management Group. *UML Profile for Enterprise Distributed Object Computing Specification*, 2002. `http://www.omg.org/cgi-bin/doc?ptc/2002-02-05`.

[21] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.