Model Driven Security: from UML Models to Access Control Infrastructures

DAVID BASIN and JÜRGEN DOSER
ETH Zurich
and
TORSTEN LODDERSTEDT
Interactive Objects Software GmbH

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 general 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.

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

General Terms: Design, Languages, Security
Additional Key Words and Phrases: Role-Based Access Control, Model Driven Architecture, Unified Modeling Language, Object Constraint Language, metamodeling, security engineering

This work has been supported by the German "Federal Ministry of Economics and Labor" under the reference number IT-MM-01MS107. The second author was partially supported by the Swiss "Federal Office for Education and Science" in the context of the EU-funded Integrated Project TrustCoM (IST-2002-2.3.1.9 Contract-No. 1945). The authors are responsible for the content of this publication.
Authors' addresses: D. Basin and J. Doser, ETH Zurich, 8006 Zurich, Switzerland, email: {basin,doserj}@inf.ethz.ch and T. Lodderstedt, Interactive Objects Software GmbH, Basler Straße 61, 79100 Freiburg, Germany, email: lodderstedt@io-software.com.

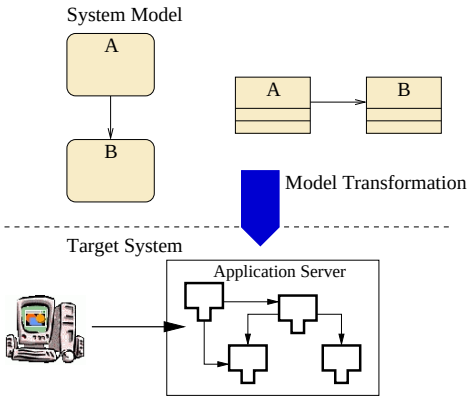

Fig. 1: Model Driven Architecture

1. INTRODUCTION

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

Model building is also carried out in security modeling and policy specification. However its integration into the overall development process is problematic and suffers from two gaps. First, security models 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 [Rumbaugh et al. 1998]). In general, the integration of system design models with security models is poorly understood and inadequately supported by modern software development processes and tools. Second, although security requirements and threats are often considered during the early development phases (requirements analysis), and security mechanisms are later employed in the final development phases (system integration and test), there is a gap in the middle. As a result, security is typically integrated into systems in a post-hoc manner, which degrades the security and maintainability of the resulting systems.

In this paper, we take up the challenge of providing languages, methods, and tools for bridging these gaps. Our starting point is the concept of *Model Driven Architecture* (MDA) [Frankel 2003], which has been proposed as 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: developers create (1) system models in high-level modeling languages like UML; tools are used to perform (2) 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 gen-

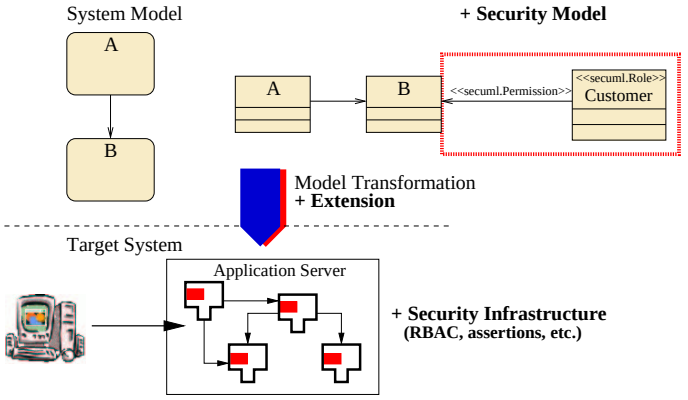

Fig. 2: Model Driven Security

eration of distributable components from class diagrams, including database access and transaction management, 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) [Ferraiolo et al. 2001]. To support visual modeling, we embed this language within an extension of UML and hence we call the result *SecureUML*. Afterwards, we give two examples of design modeling languages, one based on class diagrams and the other based on statecharts. We then combine each of these with SecureUML by defining dialects that identify particular elements of each design modeling language as protected SecureUML resources.

In each case, we define model transformations for the combined modeling language by augmenting model transformations for the UML-based modeling languages with the additional functionality necessary for translating our security modeling constructs. The first dialect provides a language for modeling access control in a distributed object setting and we define 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 [Hubert 2001] we have built a prototypical generator that implements the above mentioned transformation functions for both dialects. We report on this, as well as on experience with our approach. Overall, we view the result as a large step towards integrating security engineering into a model-driven software development process. This bridges the gap between security analysis and the integration of security mechanisms into end systems. Moreover, it integrates security models with system design models and yields a new kind of model, *security design models*.

The Model Driven Security approach that we propose offers additional advantages. 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 these transformation functions. 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 that are consistent with the formalized security policy. Third, UML is a widely-used language that developers are familiar with and many tools are available for processing UML models. Since our approach is based on UML-notation, both for defining modeling languages as well as for defining models, we expect a low acceptance barrier for our approach. Finally, we have focused on security and design modeling languages that can be given a formal semantics, both alone and in combination. This means that 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 discuss practical experience with our approach and we evaluate its generality. In Section 9.3 we review related work and in Section 10 we draw conclusions and discuss 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) 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) [Rumbaugh et al. 1998] is a widely used graphical language for modeling object-oriented systems. The language specification differentiates between *abstract syntax* and *notation* (also called *concrete syntax*). The abstract syntax defines the language primitives used to build models, whereas the notation defines the graphical representation of these primitives as icons, strings, or 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

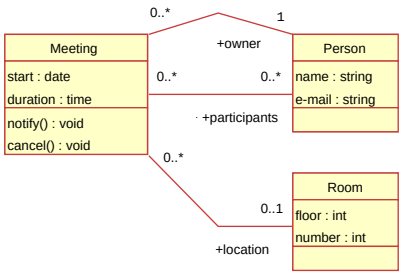

Fig. 3: Scheduler Application Class Diagram

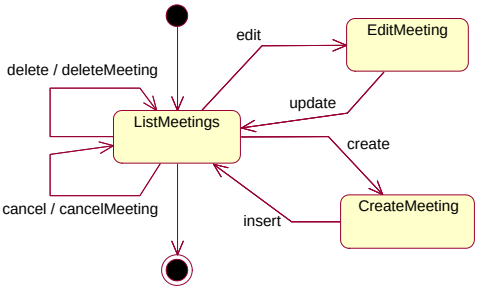

Fig. 4: Scheduler Application Statechart

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

Statemachines describe the behavior of a system or a class in terms of states and 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 the 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 *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 names a class, here Meeting, and defines an invariant of the class 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 the named class, and the reserved symbol self is used to refer to that instance. In our example, self represents an instance of the class Meeting. The attributes, association ends, and methods of an instance can be accessed either using "dot-notation" or using "arrow-notation" (in the case of operations on collections, as above). In the example, participants and owner denote the respective association ends of the meeting, and includes is a generic operation defined for arbitrary collections.

UML can 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*, formalize 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 the Model Driven Architecture (MDA) as an approach to specifying and developing applications based on platform-independent system models. Platform-specific details are incorporated into the development process by generating platform-specific models or, as in our work, even generating executable code directly from the platform-independent models. This is a large step beyond the traditional approach of using (UML) system models only for documentation and discussion purposes, or as a guideline for implementors.

One of the goals of MDA is to automate as much of the generation process as possible. Of course, the dream of automatically synthesizing complex systems from high-level descriptions is one of the holy grails of software engineering, and is unobtainable in its full generality. In particular, we cannot, in general, automatically generate the functions implementing a specification of a system's functional behavior, i.e., its "business logic". But what is possible is to automate the generation of platform-specific support for different kinds of non-functional system concerns, such as support for persistence, logging, and the like, i.e., system *aspects*, in the

aspect-oriented programming sense [Kiczales et al. 1997], that cut across different system components. Our work shows that security, in particular access control, is one such aspect that can be automatically generated in an MDA setting, and that this brings with it many advantages.

The use of domain-specific languages is at the heart of MDA, e.g., modeling languages capable of formalizing different business domains (like health care), system aspects (such as security), or concrete technologies (like Enterprise JavaBeans). There are three possibilities for defining such languages. First, domain-specific languages can be defined directly in UML in a lightweight way, using stereotypes and tagged values, as just explained in Section 2.2. Second, the OMG Meta-Object Facility (MOF) can be used to directly extend the UML metamodel (see [Frankel 2003] or the Appendix for background on MOF). A general drawback of this approach is that the customized metamodel is based on the entire UML metamodel, which is quite complex. Moreover, such a heavyweight extension may require one to extend the CASE-tool itself, in particular the storage components, i.e., the *repository*, and the visualization components. The third alternative, which we will pursue, is to define new modeling languages directly using MOF that focus on a particular problem or domain without any dependency on UML. The resulting language definitions usually have an intuitive domain-specific vocabulary that is much more concise than the vocabulary of UML-based languages. Moreover, the interfaces for querying and manipulating the metadata are simpler than a heavyweight UML interface.

As we explain in Section 3.1, we use MOF to define the *metamodels* of the modeling languages, which fix the modeling languages' abstract syntax, and afterwards we endow the languages with a concrete syntax based on UML. As is the case with the abstract syntax, there are different ways to define the concrete syntax of the new modeling language. First, it is possible to define a UML profile and introduce the concrete syntax in a UML-based CASE-tool in this way. The resulting models are then interchangeable among CASE-tools. However, this kind of notation is limited as it can only use standard UML primitives. As an alternative, dedicated modeling tools can be created for the new language. This approach is more powerful than using profiles but it also requires a higher implementation effort. Combinations of both approaches are possible: in our prototype tool, we use UML's data format for storing models and we additionally provide a custom notation for defining access control permissions.

The modeling framework of MOF plays an important role in our work, and in MDA in general. MOF is essentially a subset of UML that is used to formalize metamodels using standard object-oriented concepts like class and inheritance. A *metamodel* is a model that describes a class of models. For instance, a model m_2 can be the metamodel of another model m_1, which means that each *metaobject* (i.e., an element of a metamodel) in m_2 is a description of a type of objects in m_1 and each object in m_1 is an instance of a metaobject in m_2. In this way, metamodels specify the vocabulary that can be used to define other models. The Appendix describes MOF in more detail.

The use of MOF to define modeling languages turned out to work very well with our approach. MOF provides a more expressive formalism for defining modeling languages than lightweight UML extensions or conventional definition techniques

like the Backus-Naur Form (BNF). For example, in MOF, we can directly formalize relations between model primitives, which is one of the key ideas we use when combining modeling languages (e.g., see the discussion on subtyping in 5.1). MOF also offers advantages for building MDA tools. There is tool support for automatically creating repositories and maintaining metadata based on MOF, e.g. [Akehurst and Kent 2002]. Moreover, by separating the abstract syntax of modeling languages from their UML-based concrete syntax, we can define languages in a concise and uncluttered way and directly use UML CASE-tools for building models.

2.4 RBAC

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

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

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

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

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

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

where the symbol "$\circ$" denotes relational composition. In other words, AC is defined by

$$AC = \{(u, p) \in \mathit{Users} \times \mathit{Permissions} \mid \exists \mathit{role} \in \mathit{Roles}.\, (u, \mathit{role}) \in UA \wedge (\mathit{role}, p) \in PA\}\,.$$

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

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

where in addition to the user assignment and permission assignment relations, the role hierarchy relation $\geq$ is also part of the composition. To express the same access control relation without a role hierarchy, one must, for example, assign each user additional roles, i.e., a user is then not just assigned his original roles, but also all smaller roles. Alternatively, one can give roles additional permissions, i.e., a role not only has its assigned permissions, but also all the 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 a foundation of our security modeling language because it is well-established and it is supported by many existing technology platforms, which simplifies the subsequent definition of the transformation functions. However, RBAC also has limitations. For example, it is difficult to formalize access control policies that depend on dynamic aspects of the system, like the date or the values of system or method parameters. We have extended RBAC with authorization constraints to overcome this limitation. Furthermore, although many technologies support RBAC, they differ in details, like the degree of support for role-hierarchies and the types of protected resources. As we will see later, our approach of generating architectures from models provides a means to overcome such limitations and differences in technologies.

2.5 Security Architectures

We use three different security architectures as examples of target platforms in this paper. We provide an overview of them here, focusing in particular on their support for access control.

Enterprise JavaBeans. Enterprise JavaBeans (EJBs) is a component architecture standard [Monson-Haefel 2001] 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.

EJB differentiates between three types of components: entity beans, session beans, and message-driven beans. We will focus on the first type of components, which represent persistent business objects. The core of an EJB entity component is the *bean class*, which contains the business logic of the component. An entity component may have up to four interfaces, which can be categorized along two dimensions. First, there are *home* and *component interfaces*. Home interfaces are used, for example, to create and find bean instances, whereas component interfaces define the methods applicable to the component instances. Second, such an interface can either be a *remote* or a *local* interface. Remote interfaces can be accessed from outside the application server process, whereas local interfaces are only accessible within the Java virtual machine where the component resides.

The access control model[1] of EJB is based on RBAC, where the protected resources are the methods accessed using the interfaces of an EJB. This provides a mechanism for *declarative access control* where the access control policy is configured in the deployment descriptors of an EJB component. The security subsystem

[1]Note that the usage of the term *model* in the security community is analogous to the usage of *metamodel* as in Section 2.3. The access control model describes the language for formulating access control policies.

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`. An EJB method is identified by the name of its EJB component (`ejb-name`), the type of the interface it belongs to (`method-intf`, with the possible values `Remote`, `Local`, `Home`, and `LocalHome`), and the method signature (`method-name` and `method-params`). The listed roles are granted the right to execute the listed methods.

In general, the information needed to specify a comprehensive access control policy for realistic applications is quite voluminous. Even using good security administrative tools there is the danger that developers introduce errors due to oversights or unjustified simplifications. 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 simply define just one method permission that grants the role full access to all methods of the EJB (which can be achieved using the wild-card "*" as the method-name). The example discussed in Section 10 gives some insight into the magnitude of this problem. Model Driven Security provides a promising solution to this problem by providing a technology for modeling security policy at a high abstraction level and automatically generating the related deployment descriptors.

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.

Enterprise Services for .NET. The Microsoft Enterprise Services for .NET support the development 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 [Beyer 2001]. Here, programmatic access control allows one to obtain the identity of the caller of a method and to check the caller's role assignments. Declarative

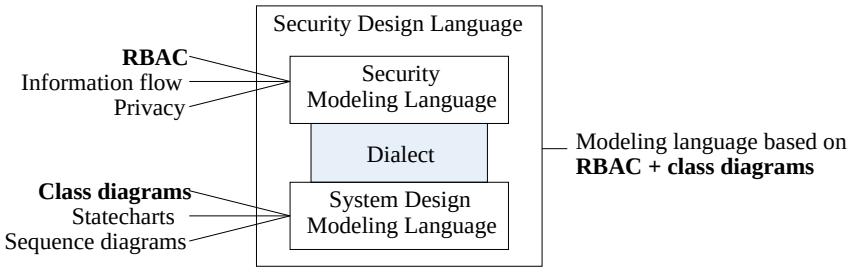

Fig. 5: Security Design Language Schema

access control 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, to an interface, 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(){...}
```

Java Servlets. The Java Servlet Specification [Hunter 2001] defines requirements on 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 both declarative and programmatic access control. For declarative access control, permissions are defined at the level of uniform resource locators (URLs) in XML deployment descriptors. Programmatic access control is used to determine the identity and the roles of a caller and to implement decisions within a servlet.

3. MODEL DRIVEN SECURITY: AN OVERVIEW

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 5. 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.

One can use standard UML as the design modeling language. As an observation, note that this is rarely done in the MDA community. The main reason is that UML is too unspecific for most design domains (technical or business). For example in most systems, there are several kinds of components or classes. One could e.g. differentiate between persistent and service components. Some business modeling frameworks distinguish primitives for defining organizations, resources, and processes. Different ways of defining domain-specific modeling languages in an MDA environment are described in Section 2.3, together with their respective advantages and disadvantages.

To automate our approach to Model Driven Security, for each schema instance we must define, as depicted in Figure 2, transformation functions that map 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 one particular 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*. Having a well-defined syntax is important for tool-developers in order to build tools for building and manipulating models in these languages. Having a well-defined semantics is important for developers and security experts to use and understand models in these languages. 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, and following UML (cf. Section 2.2), we call the underlying syntax the *abstract syntax* and the notation the *concrete syntax*. In general, the abstract syntax is 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.

An additional requirement, necessary for MDA, is that the semantics should be implementable. By this we mean that it must be possible to design a transformation function, mapping language constructs to platform specific constructs, in a way that preserves the language's semantics.

Designing a modeling language that fulfills these requirements is a creative and non-trivial task. However, it is not our expectation that each application developer must also be a language designer. This task will be done once and for all for a large class of applications by security and system architects; we examine these method-

ological questions in detail in Section 3.4. We will use SecureUML to illustrate that it is possible to design security modeling languages that are general, usable with different design modeling languages, and applicable to a wide scope of problems.

The definition of a language's syntax follows the standard approach taken in MDA, as 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. We also explain how to combine the semantics of SecureUML with those of design modeling languages.

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 levels of notation and abstract syntax. But more is required: it must be possible to build expressions in the combined language that combine subexpressions from the different languages. That is, security policies expressed in the security modeling language must be able to make statements about system resources or attributes specified in the design modeling language. It is the role of the dialect to make this connection. We will show one way of doing this based on using subtyping (in the object-oriented sense) to classify constructs in one language as belonging to subtypes in the other. We will provide examples of such combinations in Section 5 and Section 8.

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

modeling language in a way that allows the direct annotation of model elements representing protected resources with access control information. Hence it provides the missing vocabulary to formulate security policies involving these resources by defining:

—the model element types of the system design modeling language that represent protected resources;

—the actions these resource types offer and hierarchies classifying these actions;

—the rules for inheritance between resources; and

—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 5, we must define a transformation function operating on models constructed in the language. As our focus in this paper is on security, we shall assume that the system design modeling language used is already equipped with a transformation function, consisting of transformation rules that define how model elements are transformed into code or system infrastructure (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 is neither to develop nor to generate 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 respects the semantics of the security modeling language, i.e., the transformation function must be *semantic preserving*. Also, we require that the new and adapted transformation rules do not "break" the existing rules. Intuitively, this means that the system behavior should not be changed, unless where it is mandated by the semantics of the security modeling language. In the case of SecureUML, where we expect the system to have a state-transition semantics (see Section 4.3), this can be stated more precisely. Namely, we require that the transitions that are allowed by SecureUML semantics have the same effect on the system state as before adding the new transformation rules.

To judge the correctness of a transformation function requires a formal semantics for the targeted security architectures.[2] In our 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

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

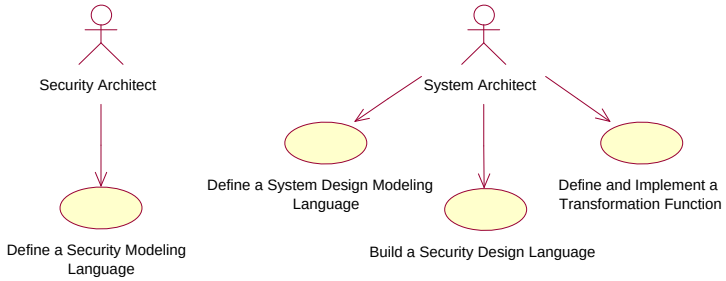

Fig. 6: The Roles and Activities in the Tool Development Process

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 could be used to 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 the 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 8.5).

3.4 Methodology

Our methodology consists of the activities carried out in two different kinds of processes. To start with, there are the activities comprising the development of security design modeling languages and accompanying tools. Such activities are carried out in a *tool development process*. The results of this process are used by software engineers in a *system development process* to design and implement secure systems.

The Tool Development Process. The UML use case diagram in Figure 6 shows the roles and activities of the tool development process. The security architect is someone knowledgeable about security and additionally has some understanding of UML and metamodeling. In contrast, a system architect should understand metamodeling techniques as well as the system domain. The responsibilities of the

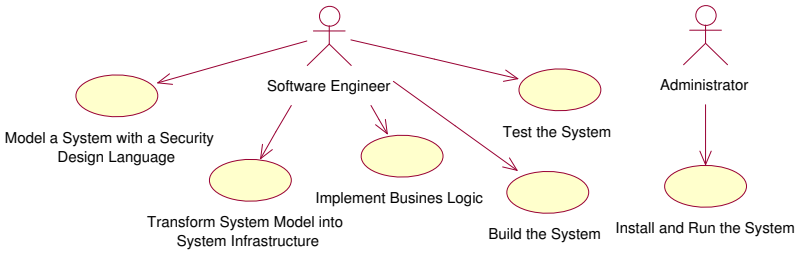

Fig. 7: The Roles and Activities in the System Development Process

security and the system architects are depicted in Figure 6.

Note that these activities must only be carried out once for a particular security and system design language. The same holds for defining and implementing the transformation function. The languages and tools can then be used by software engineers to model and construct a wide class of systems in the application domain.

The System Development Process. The system development process is characterized by the roles and activities presented in Figure 7. The difference to a standard model driven development process lies in the modeling languages used for modeling the system. Here the software engineers use modeling languages that are equipped with a vocabulary for specifying security properties of the system and the generators used to transform models into systems also create security architectures. Hence software engineers only need to understand the concepts in the UML-based security design language. They need not have expertise in the target technology or understand how the concepts are realized in the target technology.

4. SecureUML

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

4.1 Abstract Syntax

Figure 8 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 and groups to roles by using their common supertype Subject. The right-hand part of the diagram factors permissions into the ability to carry out *actions* on *resources*. These permissions may be constrained 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 actions.

Let us now examine these types and associations in more detail. Subject is the base type of all users and groups in a system. It is an abstract type (type names in *italic font* in class diagrams represent abstract types), which means that it cannot

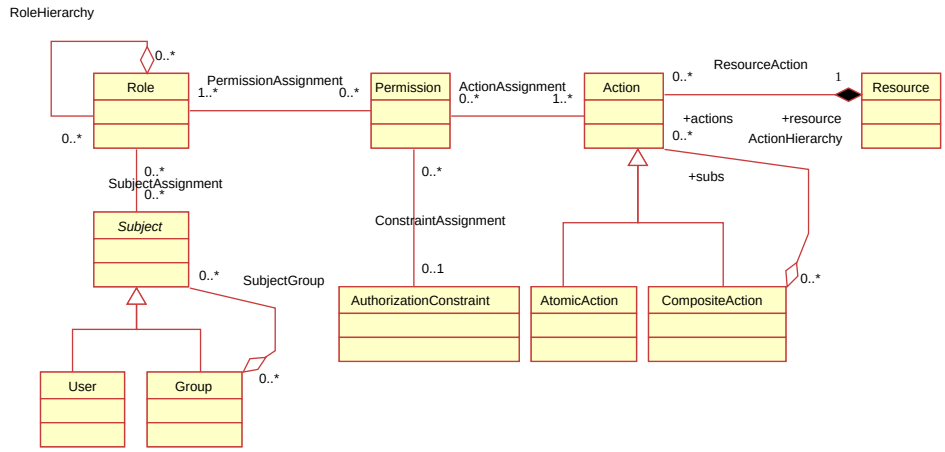

Fig. 8: SecureUML Metamodel

be instantiated directly: each subject is either a user or a group. A User represents a system entity, like a person or a process, whereas a Group names a set of users and groups. Subjects are assigned to groups by the aggregation SubjectGroup, which represents an ordering relation over subjects. Subjects are assigned to roles by the association SubjectAssignment.

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

An AuthorizationConstraint is a logical predicate that is attached to a permission by the association ConstraintAssignment and makes the permission's validity a function of the system state, 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-

Table I. Mapping between SecureUML concrete and abstract syntax

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

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 any one of the (transitively) contained subactions. This semantics yields a simple basis for defining high-level actions. Suppose that a security policy grants a role the permission to "read" an entity. Using an action hierarchy, we can formalize this by stating that such a permission includes the permission to read the value of every entity attribute and to execute every side-effect free method of the entity. 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.

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

4.2 Concrete Syntax

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

Table I 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, and ActionHierarchy); 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 9, which formalizes the second part of the security policy introduced in Section 2.1: only the owner of a meeting may change meeting data and cancel or delete the meeting. This example will allow

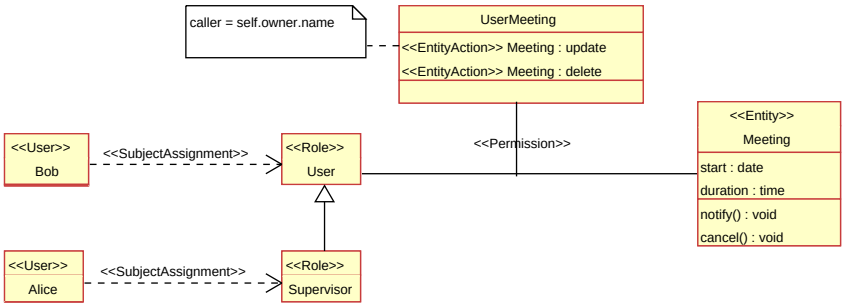

Fig. 9: Example of the concrete syntax of SecureUML

us to explain some of the fine points of our concrete syntax.

In the SecureUML profile, a role is represented by a UML class with the stereotype «Role» and an inheritance relationship between two roles is defined using a UML generalization relationship. The role referenced by the arrowhead of the generalization relationship is considered 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.

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 9 specifies a permission on a protected resource. Specifying this is only possible after having combined SecureUML with an appropriate design modeling language. The concrete syntax of SecureUML is generic in that every UML model element type can represent a protected resource. Examples are classes, attributes, and methods, as well as 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., the class Meeting is a protected resource.

A permission, along with its relations to roles (PermissionAssignment) and actions

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

Also, the reader should be careful not to confuse the role User with the SecureUML type User.

(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. Moreover, 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. The association class connects a role with a UML class representing a protected resource, which is designated as the *root resource* of the permission. The actions such a permission refers to may be actions on the root resource or on subresources of the root resource. In our example, the class Meeting is the root resource of the permission 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. "update". The resource name is stored in the tagged value identifier and references the root resource or one of its subresources. The format of the identifier depends on the type of the referenced resource and is determined by the stereotype of the attribute.

The stereotypes for action references and the naming conventions for identifiers are defined as part of the dialect. As a general rule, the resource identifier is always specified relative to the root resource. This prevents redundant information in the model and inconsistencies when the root resource's name is changed. For example, the attribute start would be referenced by the string "start" and the root resource itself would be referenced by an empty string. Note that the name of the action reference attribute has only an illustrative meaning. We generally use names that provide information about the referenced resource. In our example, the attribute of type "update" with the stereotype «EntityAction» and the name "Meeting" denotes the action *update* on the class Meeting. As we will later see in Table II, the permission to *update* an Entity also comprises the permission to *execute* any non-side-effect free method of the Entity, for example the method `cancel()` of the class Meeting. The second attribute in our example denotes the action *delete* on the class Meeting. Together, these two attributes specify the permission to update (which includes canceling) and delete a meeting.

Each authorization constraint is stored as an OCL expression in the tagged value constraint of the permission that it constrains. To improve the readability of a model, we attach a text note with the constraint expression to the permission's association class. In our example, the permission UserMeeting is constrained by the authorization constraint

caller = self.owner.name ,

which restricts the permission to update and delete a meeting to the owner of the meeting.

4.3 Semantics

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

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

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

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

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

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

To combine both RBAC configurations and authorization constraints, we combine the first-order structures $\mathfrak{S}_{St}$ and $\mathfrak{S}_{RBAC}$, as well as the first-order formulas $\phi_{St}(u, p)$ and $\phi_{RBAC}(u, a)$. Roughly speaking, the combined semantics is defined by $\langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{St} \rangle \models \phi_{AC}(u, a)$, where $\phi_{AC}(u, a)$ is built from both $\phi_{St}(u, p)$ and $\phi_{RBAC}(u, a)$, and $\langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{St} \rangle$ denotes the "union" of the structures $\mathfrak{S}_{RBAC}$ and $\mathfrak{S}_{St}$. Of course, the addition of access control changes the behavior of the system. We therefore must also define how the semantics of SecureUML changes the behavior specified by the design modeling language. To accomplish this, we represent the system behavior as a transition system and interpret the addition of access control as restricting system behavior by removing transitions from this transition system. In the following paragraphs we define these intuitive concepts more precisely.

Declarative Access Control. First, we define an order-sorted signature Σ_{RBAC} that defines the type of structures that specify role-based access control configurations.[4] We define this signature by $\Sigma_{RBAC} = (\mathcal{S}_{RBAC}, \mathcal{F}_{RBAC}, \mathcal{P}_{RBAC})$, where $\mathcal{S}_{RBAC}$ is a set of sorts, $\mathcal{F}_{RBAC}$ is a set of function symbols, and $\mathcal{P}_{RBAC}$ is a set of predicate symbols. In detail, we define

$$\begin{aligned} \mathcal{S}_{RBAC} &= \{ \mathit{Users}, \mathit{Subjects}, \mathit{Roles}, \mathit{Permissions}, \mathit{Actions} \} \ , \\ \mathcal{F}_{RBAC} &= \emptyset \ , \\ \mathcal{P}_{RBAC} &= \{ \geq_{\mathit{Subjects}}, \mathit{UA}, \geq_{\mathit{Roles}}, \mathit{PA}, \mathit{AA}, \geq_{\mathit{Actions}} \} \ , \end{aligned}$$

where *Users* is a subsort of *Subjects*, $\geq_{Subjects}$ has type $Subjects \times Subjects$, UA has type $Subjects \times Roles$, $\geq_{Roles}$ has type $Roles \times Roles$, PA has type $Roles \times Permissions$, AA has type $Permissions \times Actions$, and $\geq_{Actions}$ has type $Actions \times Actions$. The

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

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

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

Additionally, we define the partial orders $\geq_{Subjects}$, $\geq_{Roles}$, and $\geq_{Actions}$ on the sets of subjects, roles, and actions respectively. $\geq_{Subjects}$ is given by the reflexive closure of the aggregation association SubjectGroup in Figure 8 and formalizes that a group is larger than all its contained subjects. $\geq_{Role}$ is defined analogously based on the aggregation association RoleHierarchy on Role and we write subroles (roles with additional privileges) on the left (larger) side of the $\geq$-symbol. $\geq_{Actions}$ is given by the reflexive closure of the composition hierarchy on actions, defined by the aggregation ActionHierarchy. We write $a_1 \geq_{Actions} a_2$, if a_2 is a sub-action of a_1. These relations are partial orders because aggregations in UML are transitive and antisymmetric by definition.

Note that compared to Figure 8, we have deliberately excluded the metamodel types Group and Resource. Resource is excluded because the target of access control is the actions performed on resources, and not resources themselves. Group is excluded because groups are just a subset of subjects and do not play any further role in the semantics.

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

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

or equivalently, by factoring out the permissions explicitly, as

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

where

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

states that the user u has the permission p, and

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

states that p is a permission for the action a. This is essentially a reformulation of the usual RBAC semantics (cf. Section 2.4). The use of definition (1) will become

clear when we combine this formula with programmatic access control formulas ϕ_{St}.

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

Programmatic Access Control. While declarative access control decisions can be made without referring to the system model, we need to explicitly incorporate the syntax and semantics of the design modeling language into SecureUML to make programmatic access control decisions. In order to be able to combine the semantics of SecureUML with the semantics of system design modeling languages, we make some assumptions about the nature of the latter, so that a combination of the semantics can be well-defined.

For making programmatic access control decision, we require that the system design modeling language provides a vocabulary for talking about the structure of the system. More formally, we require that the system design modeling language provides an order-sorted first-order signature $\Sigma_{St} = (\mathcal{S}_{St}, \mathcal{F}_{St}, \mathcal{P}_{St})$, where $\mathcal{S}_{St}$ is a set of sorts, $\mathcal{F}_{St}$ is a set of typed function symbols (including constants), and $\mathcal{P}_{St}$ is a set of typed predicate symbols. Typically, $\mathcal{S}_{St}$ contains one sort for each class in the system model, $\mathcal{F}_{St}$ contains a function symbol for each attribute and for each side-effect free method of the model, and $\mathcal{P}_{St}$ contains predicate symbols for 1-to-many and many-to-many relations between classes. How exactly this signature is defined depends on the semantics of the system design modeling language. We do however require that $\mathcal{F}_{St}$ contains a constant symbol $self_C$ for each class C in the system model. Also, how the symbols in Σ_{St} are interpreted in $\mathfrak{S}_{St}$ is again defined by the system design modeling language. Here we require that the constant symbol $self_C$ is interpreted by the currently accessed object, in case the currently accessed object is of the sort C.

In this setting, the state of the system at a particular time defines a Σ_{St}-structure $\mathfrak{S}_{St}$. Constraints on the system state $\mathfrak{S}_{St}$ can be expressed as logical formulas ϕ_{St}, whereby satisfaction of constraints is just the question of whether $\mathfrak{S}_{St} \models \phi_{St}$ holds.[5]

Combining Declarative and Programmatic Access Control. To formalize combined declarative and programmatic access control decisions, we combine the states $\mathfrak{S}_{St}$ and $\mathfrak{S}_{RBAC}$ into the composite structure $\mathfrak{S}_{AC} = \langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{St} \rangle$ and combine the formulas ϕ_{St} and ϕ_{RBAC} to a new formula ϕ_{AC}. The combined access control decision is then defined by the question of whether $\mathfrak{S}_{AC} \models \phi_{AC}$ holds. By $\langle \mathfrak{S}_{RBAC}, \mathfrak{S}_{St} \rangle$ we mean that $\mathfrak{S}_{AC}$ is the structure that consists of the carriers sets, functions and predicates from both $\mathfrak{S}_{RBAC}$ and $\mathfrak{S}_{St}$. As for ϕ_{AC}, in the simplest case it is just the conjunction of ϕ_{RBAC} and ϕ_{St}, i.e., $\phi_{AC} = \phi_{RBAC} \wedge \phi_{St}$, stating that both the declarative and the programmatic access control must grant access.

For realistic security policies, ϕ_{AC} is usually more than just a simple conjunction. First, authorization constraints are not global constraints, but are attached to permissions, for example as in Figure 9. This means, authorization constraints are only relevant for the roles that have these respective permissions. Therefore, we

[5]As explained previously, authorization constraints are OCL formulas. Note that there is a straightforward translation from OCL constraints without collections into the language of first-order logic. [Beckert et al. 2002] describes such a translation as well as the less straightforward issue of how to translate OCL collections.

have to refer to the internal structure of ϕ_{RBAC} when combining ϕ_{RBAC} with ϕ_{St}. Second, we would like to refer to parts of the RBAC configuration while expressing authorization constraints, in particular to the user. This allows one to formalize constraints (also as in Figure 9) like "the current caller is the owner of the meeting." Therefore, we combine the signatures Σ_{RBAC} and Σ_{St} by taking their union, i.e., $\Sigma_{AC} = (\mathcal{S}_{AC}, \mathcal{F}_{AC}, \mathcal{P}_{AC})$, where $\mathcal{S}_{AC} = \mathcal{S}_{RBAC} \cup \mathcal{S}_{St}$, $\mathcal{F}_{AC} = \mathcal{F}_{RBAC} \cup \mathcal{F}_{St}$, and $\mathcal{P}_{AC} = \mathcal{P}_{RBAC} \cup \mathcal{P}_{St}$, and we require that authorization constraints ϕ_{St} are expressions in the first-order language over this signature Σ_{AC}. Observe that under this definition of Σ_{AC}, $\mathfrak{S}_{AC}$ is a Σ_{AC}-structure.

The combined access control semantics is now defined by

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

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

Behavioral Semantics of Access Control. The preceding paragraphs defined how access control decisions are made in a system state. But what is interesting, in the end, is how the system behaves when an access control decision is made. To be able to define this precisely, we again make some minimal assumptions on the semantics of the design modeling language. Namely, we assume that the semantics of the system design modeling languages can be abstracted to a labeled transition system (LTS) $\Delta = (Q, A, \delta)$. In this LTS, the set of nodes Q consists of Σ_{St}-structures, the edges are labeled with elements from a set of actions A, and $\delta \subseteq Q \times A \times Q$ is the transition relation. The behavior of the system is defined by the paths (also called traces) in the LTS as is standard: a trace $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_{n+1}$ defines a possible behavior if and only if $(s_i, a_i, s_{i+1}) \in \delta$, for $0 \leq i \leq n$.

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

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

—Q_{AC} is defined by combining system states with RBAC configurations, i.e., $Q_{AC} = Q_{RBAC} \times Q$, where Q_{RBAC} denotes the universe of all finite Σ_{RBAC}-structures. We define $\pi_{St} : Q_{AC} \to Q$ and $\pi_{RBAC} : Q_{AC} \to Q_{RBAC}$ to be the respective projections.

—$A_{AC} = A$ is unchanged.

—δ_{AC} is defined by restricting the transitions from δ and lifting them to Q_{AC}:

$$\delta_{AC} = \{(q, a, q') \in \mathit{lift}(\delta) \mid q \models \phi_{AC}(u,a)\} \ ,$$

where $\mathit{lift}(\delta)$ denotes the lifting of δ to Q_{AC}, i.e.,

$$\mathit{lift}(\delta) = \left\{ (q, a, q') \in Q_{AC} \times A_{AC} \times Q_{AC} \mid \begin{array}{c} (\pi_{St}(q), a, \pi_{St}(q')) \in \delta \wedge \\ \pi_{RBAC}(q) = \pi_{RBAC}(q') \end{array} \right\} \ .$$

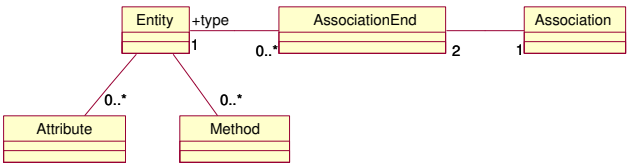

Fig. 10: ComponentUML Metamodel

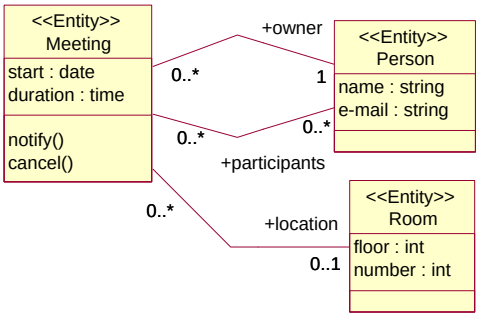

Fig. 11: Scheduling application

Note that this definition of $lift(\delta)$ also ensures that the RBAC configuration does not change during system execution, i.e., it really is static.

We will see concrete semantic combinations of SecureUML with different design modeling languages in Sections 5.3 (for ComponentUML) and in Section 8.3 (for ControllerUML).

5. AN EXAMPLE MODELING LANGUAGE: ComponentUML

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

ComponentUML is a simple language for modeling distributed object-oriented systems. The metamodel for ComponentUML is shown in Figure 10. Elements of type Entity represent business object types of a particular domain. An entity may have multiple methods and attributes, represented by elements of the types Method and Attribute respectively. Associations are used to specify relations between entities. An association is built from an Association model element and every entity participating in an association is connected to the association by an AssociationEnd.

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

Figure 11 shows the structural model of our scheduling application in the ComponentUML notation. Instead of classes, we now have the three entities Meeting, Person, and Room, each represented by a UML class with the stereotype «Entity».

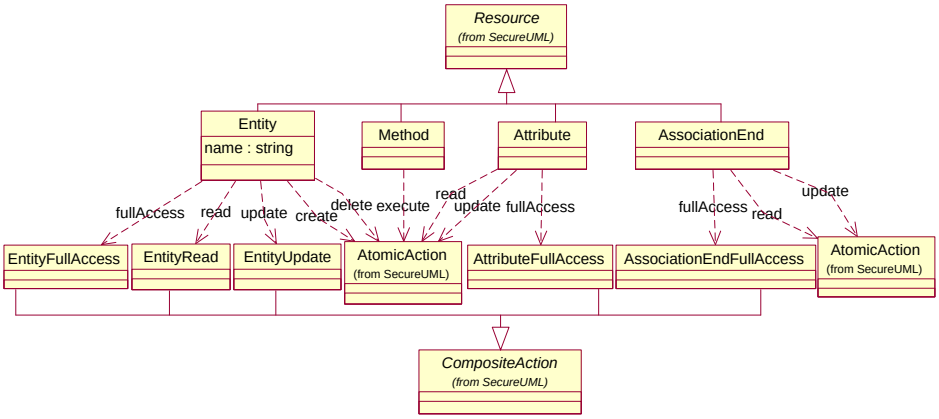

Fig. 12: SecureUML Dialect for ComponentUML Metamodel

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 achieved by importing the SecureUML metamodel into the metamodel of ComponentUML. This extends ComponentUML with the SecureUML modeling constructs, e.g., Role and Permission. The use of packages and corresponding namespaces for defining these metamodels ensures that no conflicts arise during merging. While this step can be automated, the following steps cannot be as they are creative tasks.

Identifying Protected Resources. Second, we identify the model elements of ComponentUML representing protected resources and formalize this as part of a SecureUML dialect. To do this, we must determine which model element we wish to control access to in the resulting systems. Moreover, when doing this, we must account for what can ultimately be protected by the target platform. Suppose, for example, we decide to interpret entity attributes as protected resources and the target platform supports access control on methods only. This is possible, but it necessitates a transformation function that transforms each modeled attribute into a private attribute and generates (and enforces access to) access methods for reading and changing the value of the attribute in the generated system.

In our example, we identify the following model elements of ComponentUML as protected resources: Entity, Method, Attribute, and AssociationEnd. This identification is realized by letting these metatypes inherit from the SecureUML type Resource as shown in Figure 12. In this way, the metatypes inherit all properties needed to define authorization policies. Additionally, we define in this figure several action classes as subtypes of the SecureUML class CompositeAction. The action composition hierarchy can then be defined as part of each action's type information, by way of OCL invariant constraints (see below) on the respective types.

Defining Resource Actions. In the next step, we define the set of actions that is offered by every model element type representing a protected resource, i.e. we

Table II. SecureUML dialect action hierarchy

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

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 kinds and connect them using hierarchies.

In the metamodel, the set of actions each resource type offers is defined by the named dependencies from the resource type to action classes, as shown in Figure 12. 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 12 formalizes that an Attribute always possesses the action *fullAccess* of type AttributeFullAccess and the actions *read* and *update* of type AtomicAction.

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

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(
```

Table III. Action Reference Types for ComponentUML

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

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. Here, the tagged value "isQuery" is used to select the side-effect free methods. 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 can be 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 permission, which is any UML class in the SecureUML notation (see Section 4.2), is restricted to UML classes with the stereotype «Entity». Finally, as shown in Table III, we define the action reference types for entities, attributes, methods, and association ends.

5.3 Extending the Semantics

We have to define the semantics of ComponentUML as a labeled transition system $\Delta = (Q, A, \delta)$ over a fixed first-order signature Σ_{St} (cf. Section 4.3). Intuitively, every entity defines a sort in the first-order signature, and every atomic action defined by the SecureUML dialect for ComponentUML (cf. Figure 12) defines an action in the labeled transition system. Side-effect free actions give rise to function and predicate symbols in the first-order signature.

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

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

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

—Each side-effect free entity method m whose tagged value "isQuery" is set to true gives rise to a function symbol f_m in $\mathcal{F}_{St}$ of the corresponding type. Corresponding type here means, in particular, that we add the sort of the entity as an additional parameter, i.e., the "this-pointer" is passed as an additional argument. Also, each entity attribute at gives rise to a function symbol get_{at} in $\mathcal{F}_{St}$ (the "get-method") of type $s \rightarrow v$, where s is the sort of the entity, and v is the sort of

the attribute's type. Finally, each association end ae with multiplicity $\{1\}$ gives rise to a function symbol f_{ae}:

$$\begin{aligned}\mathcal{F}_{St} = &\{f_m \mid m \text{ is an entity method}\} \cup \\ &\{get_{at} \mid at \text{ is an entity attribute}\} \cup \\ &\{self_e \mid e \text{ is an entity}\} \cup \\ &\{f_{ae} \mid ae \text{ is an association end with multiplicity } \{1\}\} \ .\end{aligned}$$

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

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

Following this, we define the labeled transition system $\Delta = (Q, A, \delta)$ by:

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

—The set of actions A is defined by:

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

(cf. Figure 12) where, for example, AttributeUpdateActions is defined by:

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

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

—The transition relation $\delta \subseteq Q \times A \times Q$ defines the allowed transitions. The exact details of δ will depend on the intended semantics of the methods themselves. We will just give a few examples here to illustrate the main idea. This should suffice, since, as should be clear from Section 4.3, our approach does not depend on the definition of δ. For example, for $a \in \mathit{AttributeReadActions}$, $(q, a, q') \in \delta$ if and only if $q = q'$, i.e., reading an attribute's value does not change the system state. In contrast, setting an attribute value should be reflected in the system state: for $a = (set_{at}, e, v) \in \mathit{AttributeUpdateActions}$, $(q, a, q') \in \delta$ implies $q' \models get_{at}(e) = v$.

Having defined the semantics of ComponentUML in this way, we combine it with the semantics of SecureUML as described in Section 4.3. This means, we define the combined transition system $\Delta_{AC} = (Q_{AC}, A_{AC}, \delta_{AC})$ by adding Σ_{RBAC}-structures

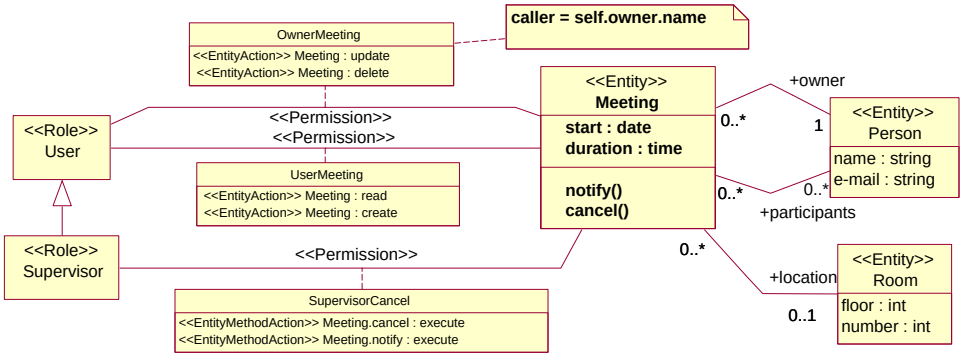

Fig. 13: Scheduler Example with Authorization Policy

to the system states in Q, lifting δ to $\mathit{lift}(\delta) \subseteq Q_{AC} \times A_{AC} \times Q_{AC}$, and removing the forbidden transitions from $\mathit{lift}(\delta)$. Hence, δ_{AC} will only contain those transitions that are allowed according to the SecureUML semantics.

5.4 Modeling the Authorization Policy

We now use the combined language to formalize the security policy given in Section 2.1. We do this by adding permissions to the entity model of the scheduler application that formalize the three policy requirements. As these permissions associate roles with actions, we also employ the roles User and Supervisor, which we introduced in Section 4.2.

The first requirement states that any user may create and read meeting data. We formalize this by the permission UserMeeting in Figure 13, 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 a 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 meeting instance. Due to the definition of the action update (cf. Table II), 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 of 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.

5.5 Examples of Access Control Decisions

We now illustrate the semantics by analyzing several access control decisions in the context of Figure 13. We assume that we have three users, Alice, Bob, and Jack, and that Bob is assigned the role `User` whereas Alice is assigned the role `Supervisor`. Here we assume that our dialect has the *default behavior* "access allowed" and we directly apply the semantics of SecureUML to the example policy

given in the previous section. The corresponding Σ_{RBAC}-structure $\mathfrak{S}_{RBAC}$ is

$$\begin{aligned}
Users &= \{\text{Bob}, \text{Alice}, \text{Jack}\} \\
Roles &= \{\texttt{User}, \texttt{Supervisor}\} \\
Permissions &= \{\texttt{OwnerMeeting}, \texttt{SupervisorCancel}, \dots\} \\
Actions^{6} &= \{\texttt{Meeting.update}, \texttt{Meeting::cancel.execute}, \dots\} \\
UA &= \{(\text{Bob}, \texttt{User}), (\text{Alice}, \texttt{Supervisor})\} \\
PA &= \{(\texttt{User}, \texttt{OwnerMeeting}), (\texttt{Supervisor}, \texttt{SupervisorCancel}), \dots\} \\
AA &= \{(\texttt{OwnerMeeting}, \texttt{Meeting.update}), \\
&\quad\ (\texttt{SupervisorCancel}, \texttt{Meeting::cancel.execute}), \dots\} \\
\geq_{Roles} &= \{(\texttt{Supervisor}, \texttt{User}), (\texttt{Supervisor}, \texttt{Supervisor}), (\texttt{User}, \texttt{User})\} \\
\geq_{Actions} &= \{(\texttt{Meeting.update}, \texttt{Meeting::cancel.execute}), \dots\} \ ,
\end{aligned}$$

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

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

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

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

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

Now suppose that Alice wants to cancel a meeting entry owned by Jack. Suppose further that the system state is given by the interpretation of the first-order structure $\mathfrak{S}_{St}$ over Σ_{St}, where

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

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

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

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

The constraint expression "caller = self.owner.name" on the permission OwnerMeeting is translated into the formula

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

and the formula for the permission SupervisorCancel is *true*. The access decision is formalized as

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

which is satisfied.

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

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

Hence,

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

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, and user assignment, we generate a corresponding element in the deployment descriptor. We therefore omit these details and focus here on the parts of the infrastructure responsible for enforcing permissions and authorization constraints.

6.1 Basic Generation Rules for EJB

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

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

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

```
  ...
</entity>
```

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

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

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

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

The transformation applied to elements of type AssociationEnd is similar to the rules defined for attributes. Two access methods are generated for reading and writing the collection of associated objects. Furthermore, persistency information for storing the association-end data in a database is generated. The following code fragment shows the declarations of the access methods for the association end participants of the entity Meeting.

```
Collection getParticipants();
void setParticipants(Person participant);
```

6.2 Generating Access Control Infrastructures

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

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

Generating Permissions. As explained in Section 2.5, a method permission element names the set of roles and the set of EJB methods that the 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

Table IV. Atomic action to method mapping for EJB

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

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 subactions (under the reflexive closure of the relation defined by the association ActionHierarchy) are added to the action set. Finally, for each atomic action in the resulting set, `method` elements for the corresponding EJB methods are generated. The correspondence between atomic actions and EJB methods is given in Table IV. 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 IV 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 14.

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.

```
<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>
```

Fig. 14: Generated XML code for the methods of the permission UserMeeting

First, given a method m, the atomic action a corresponding to the method is determined using Table IV. 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 p is determined, again taking into account the hierarchy on roles in the same way as in the previous section. Finally, based on this information, an assertion of the following form is generated

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

This scheme is similar to Equation (2) on page 25, 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 permission assignments and action assignments are known at compile time, this information is used to simplify the assertion. Instead of considering all permissions, we only consider permissions that refer to the action in question by calculating the set $ActionPermissions(a)$. This has the effect, that the equivalent to $\phi_{Action}(p, a)$ in Equation (2) can be omitted. Similarly, the equivalent to ϕ_{User} is simplified by only considering roles that have one of these permission, which is done by calculating the sets $PR(p)$. If a constraint is assigned to a permission, it is evaluated afterwards. Access denial is signaled to the caller by throwing an exception.

As an example, the following 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>)`. The variable `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 The Correctness of Generation

As stated in Section 3, judging the correctness of the transformation process requires a formal semantics for the target 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 semantics of SecureUML.

Informal Semantics of The EJB Security Architecture. In the EJB context, the protected resources are the methods of the entity beans. Each method provides the single action to "execute this method", the collection of which forms 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 the 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, the body of methods can be prefaced by code whose evaluation determines if the execution of the 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.

Sketch of the 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 been assigned an authorization constraint.

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 the execution of the relevant methods corresponding to this atomic action, which is correct.

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

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

i.e., access is allowed if and only if the user u is assigned to a role that is larger than or equal to a role that has a permission p and this permission refers to an action that is larger than or equal to the atomic action corresponding to executing this method (cf. Equation (1) 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 $\phi_{RBAC}(u,a)$ is true. 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 (i.e., $\phi_{Action}(p,a)$ holds), contains a role that the user u has if and only if $\phi_{User}(u,p)$ holds.

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

$$\bigvee_{p \in Permissions} \phi_{User}(u,p) \wedge \phi_{Action}(p,a) \wedge \phi_{St}(p) \ , \tag{5}$$

evaluates to *true*. However, looking at the definitions of ActionPermissions(a) and PR(p) one sees that $p \in \mathrm{ActionPermissions}(a)$ corresponds to $\phi_{Action}(p,a)$, and that $\bigcup_{r \in \mathrm{PR}(p)} \mathrm{UserRole}(r)$ corresponds to $\phi_{User}(u,p)$. This means that both formulas are essentially the disjunction over the same set of constraints $\phi_{St}(p)$.

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. 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). 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. 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 the 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 the application's components.

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.

```
if (!(ctxt.IsCallerInRole("Supervisor") /* SupervisorCancel */
```

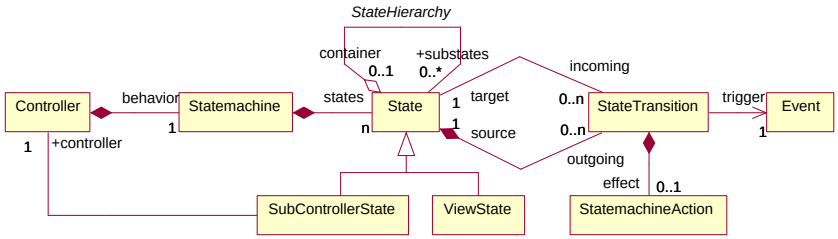

Fig. 15: Metamodel of ControllerUML

```
  || (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.[7] We will show how ControllerUML can be integrated with SecureUML and used to model secure controllers for multi-tier applications and how access control infrastructures can be generated from such controller models.

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

The abstract syntax of ControllerUML is defined by the metamodel shown in Figure 15. Each Controller possesses a Statemachine that describes its behavior in terms of States, StateTransitions, Events, and StatemachineActions. A State may contain other states, formalized by the association StateHierarchy, and a transition between two states is defined by a StateTransition, which is triggered by the event referenced by the association end trigger. A statemachine action specifies 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 humans 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, which 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 of a ControllerUML model is shown in Figure 16. A

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

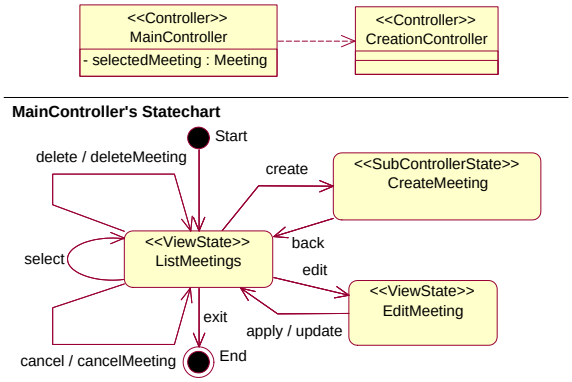

Fig. 16: Controllers for the Scheduling Application

Controller is represented by a UML class with the stereotype «Controller». The behavior of the controller is defined by a statemachine that is associated to this UML class. States, transitions, events, and actions are represented by their counterparts in the UML metamodel. Transitions are labeled with a string, containing a triggering event and an action to be executed during state transition, separated by a slash. We use events to name transitions in our explanations. View states and subcontroller states are labeled by the stereotypes «ViewState» and «SubControllerState», respectively.

Figure 16 shows the design model for an interactive application that formalizes the scheduler workflow presented in Section 2.2. The controller class MainController is the top-level controller of the application and CreationController controls the creation of new meetings (details are omitted here to save space). The 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 transition apply 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. 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. For example, one can choose whether entering states or making transitions (or both) are protected. Each way results in the definition of a different dialect for integrating ControllerUML with SecureUML. Here we shall

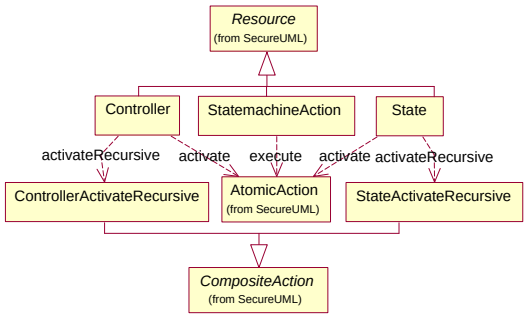

Fig. 17: Resource Model of ControllerUML

proceed by focusing on the structural aspects of statecharts, which are described by the classes of the metamodel (Figure 15) 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 17 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(
  name = "execute")->union(
  resource.substates.actions->select(name = "activateRecursive"))) .
```

This expression is built using the vocabulary defined by the ControllerUML metamodel shown in Figure 15 and the dialect definition given in Figure 17. The third line accesses the resource that the action belongs to (always a state) and selects the action with the name "activate". The next line queries all outgoing transitions on the state and selects those transitions with an assigned 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 the name "activateRecursive" on all substates of the state to which the action of type StateActivateRecursive belongs.

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

Table V. Action Reference Types for ControllerUML

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

8.2 Extending the Notation

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

8.3 Extending the Semantics

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

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

—Each Controller c gives rise to two sorts C_c and S_c in $\mathcal{S}_{St}$. C_c is the sort of the controller c, where the elements of sort C_c represent the instances of the controller c. S_c is the sort of the states of the controller c, where each state of the statemachine that describes the behavior of the controller c gives rise to an element of sort S_c. Additionally, $\mathcal{S}_{St}$ contains the sorts String, Int, Real, and Boolean:

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

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

$$\mathcal{F}_{St} = \{get_{at} \mid at \text{ is a controller attribute}\} \cup \{self_c \mid c \text{ is a controller}\} \ .$$

The initial and current states of a controller's statemachine are denoted by attributes `initialState` and `currentState` of type S_c.

—Since there are no predicate symbols,

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

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

—Q is the universe of all possible states, which is just the set of all first-order structures over the signature Σ_{St} with finitely many elements for each controller sort, where the interpretations of `String`, `Int`, `Real`, `Boolean`, `User`, and S_c are fixed to be the sets *Strings*, $\mathbb{Z}$, $\mathbb{R}$, $\{true, false\}$, and the set of states of the controller c respectively.

—The set of actions A is defined by:

$$A = \mathit{ControllerActivateActions} \cup \mathit{StateActivateActions} \cup \mathit{SMActionExecuteActions} \cup \mathit{StateTransitions} \ .$$

This means that all atomic action (cf. Figure 17) as well as all state transitions are actions of the transition system.

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

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

Note that it is possible to combine a ControllerUML model with a ComponentUML model by merging the sorts, function and predicate symbols defined by them. For example, in Figure 18 in the MainController, we refer to the entity Meeting of the ComponentUML model.

8.4 Formalizing the Authorization Policy

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

As Figure 18 shows, we use two permissions to formalize the first requirement that all users are allowed to create and to read all meetings. The permission UserMain grants the role User the right to activate the controller MainController and the states ListMeetings and CreateMeeting. The permission UserCreation grants the role User the privilege to activate the CreationController including the right to activate all of its states and to execute all of its actions.

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

$$\mathsf{caller} \ = \ \mathsf{self}\,.\,\mathsf{selectedMeeting}\,.\,\mathsf{owner.name} \ ,$$

which is attached to the permission.

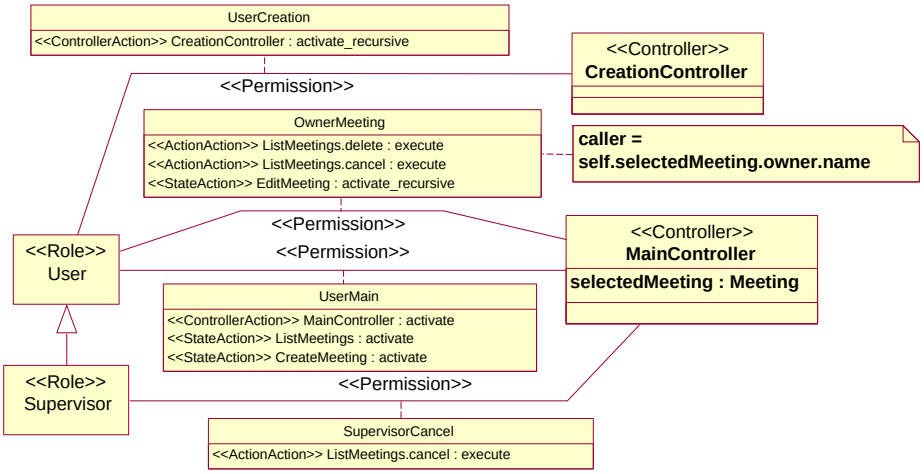

Fig. 18: Policy for Scheduling Application

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

8.5 Transformation to Web Applications

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

The Java Servlet architecture supports RBAC; however, its URL-based authorization scheme only enforces access control when a request arrives from outside the web server. This is ill-suited for 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. Afterwards, an assertion is generated of the form:

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

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

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

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

9. EVALUATION, SCOPE AND RELATED WORK

9.1 Evaluation

We have evaluated the ideas presented in this paper in an extensive case study based on a model-driven version of the J2EE "Pet Store" application, which is a prototypical application designed to demonstrate the use of the J2EE platform. Pet Store is an e-commerce application with web front-ends for shopping, administration, and order processing. The application model consists of 30 components and several front-end controllers. We have extended this model with an access control policy formalizing the principle of least privileges [Mayfield et al. 1991], where a user is only given those access rights that are necessary to perform a job. The modeled policy comprises six roles and 60 permissions, 15 of which are restricted by authorization constraints. The corresponding infrastructure is generated automatically and consists of roughly 5,000 lines of XML (overall application: 13,000) and 2,000 lines of Java source code (overall application: 20,000).

This large expansion is due to the high abstraction level provided by the modeling language. For example, we can grant a role in the model "read" access to an entity, whereas EJB only supports permissions for whole components or single methods. Therefore, a modeled permission to read the state of a component may require the generation of many method permissions, e.g., for the get-methods of all attributes. Clearly, this amount of information cannot be managed at the source code level. The low abstraction level provided by the access control mechanisms of today's middleware platforms therefore forces developers to take shortcuts and make compromises in the use of access control mechanisms. For example, roles are assigned full access privileges even where they only require read access. As another

example, access control is enforced by providing different applications for different viewpoints while the backend is only protected against external attackers by using firewalls and virtual private networks. However, this ignores the threats of internal attacks. As our experience shows, Model Driven Security cannot only help to ease the transition from security requirements to secure applications, it also plays an important role in helping system designers to formalize and meet exact application requirements.

9.2 Alternative Security Models

We have focused in this paper on a particular instance of Model Driven Security for design models combined with SecureUML. SecureUML and the generated infrastructures are based on extensions of RBAC. We believe, however, that the scope of Model Driven Security is much more general. To support this thesis, we briefly sketch here its application to two other popular access control models.

Chinese Wall. Chinese Wall policies [Brewer and Nash 1989] formalize the notions of conflict of interest classes and of being on the "wrong side of the wall". In this formalism, data is organized according to the company that controls it, and the set of companies is partitioned into conflict of interest classes. An employee may access a data object of some particular company, i.e., he is on the "right side of the wall", if either he has accessed data of this company before, or he has not previously accessed data of a different company in the same conflict of interest class. Chinese Walls intuitively capture the policy that must be adhered to by a management consultant. Such a consultant should not advise companies where he has gained insider knowledge of a competitor, i.e., where he has accessed data of a different company in the same conflict of interest class.

Modeling which company controls what data and the membership of companies in conflict of interest classes can, for example, be done by statically assigning data objects to companies and companies to conflict of interest classes, using associations comparable to the subject-role and user-group assignments of SecureUML. The transformation rules for such models must then generate book-keeping code that tracks which company's data a user has already accessed. The rules must also generate appropriate assertions that check whether the current caller is on the "right side of the wall".

Bell-LaPadula. The main idea of the Bell-LaPadula model [Bell and LaPadula 1976] is to classify objects by security levels and to grant subjects clearance for individual levels. The security levels are taken from a partially ordered set, e.g., {unclassified, confidential, secret} where unclassified $<$ confidential $<$ secret, and the clearance of a subject defines which objects he can access, depending on the type of access (read, write, or append). The primary two rules of the Bell-LaPadula model are often informally summed up by the phrases that "no read up" and "no write down" are allowed. Usually, the Bell-LaPadula model also incorporates an access control matrix, which maps subject-object pairs to allowed access types, but we ignore this here in the discussion, as this can be handled similarly to SecureUML.

Assigning security levels to data objects and subjects, as well as classifying actions into read, write, or append actions, can be done as in the case of Chinese Wall policies. The transformation rules then only have to generate assertions that check

the "no read up" and "no write down" conditions.

9.3 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 Jaeger [1999]. A formal language to express these constraints, based on first-order logic, is, for example, proposed by Chen and Sandhu [1996]. Ahn and Sandhu develop the "RSL99 language for Role-Based Separation of Duty Constraints" [Ahn and Sandhu 1999] and the Role-Based Constraint Language RCL2000 [Ahn and Sandhu 2000]. Ahn and Shin [2001] 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.

Ahn and Shin [2000] 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, Epstein and Sandhu [1999] 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, Epstein and Sandhu [1999] do not consider the question of implementing infrastructures for enforcing access control policies, whereas we propose a generative approach.

Jürjens [2001; 2002] 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 and for analyzing these policies.

Probably the most closely related work is the Ponder Specification Language [Damianou et al. 2001; Damianou 2002], which supports the formalization of authorization policies where rules specify which *actions* each *subject* can perform on given *targets*. As in our work, Ponder supports the organization of privileges in 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 for formalizing and combining modeling languages; the modeling language SecureUML for specifying access control policies, which constitutes a substantial extension of RBAC; and techniques for generating platform-specific access control 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 abstraction. What is attractive here is that our use of dialects to join languages provides a way of decomposing language design so that these problems can be tackled independently.

We 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 5.5. More complex kinds of analysis should be possible too. Future work is to investigate the possibilities for analyzing security design models. Ideas here include calculating a symbolic description of those system states where an action is allowed, model checking statechart diagrams that combine dynamic behavior specifications with security policies, and verifying refinement or consistency relationships between different models.

Finally, the question remains of how Model Driven Security can be integrated into the overall system development process. For example, how can roles and pro-

Table VI. The MOF metadata architecture

Meta level	Description	Example elements
M3	MOF Model	MOF Class, MOF Attribute
M2	Metamodel, defines a language	Entity, Attribute
M1	Model, consisting of instances of M2 elements	Entities “Meeting” and “Person”
M0	Objects and data	Persons “Alice” and “Bob”

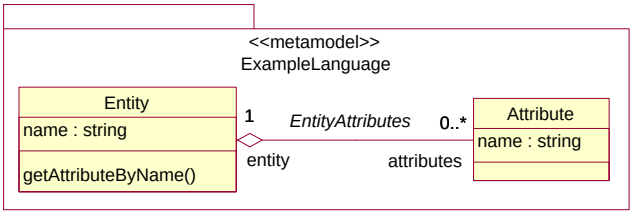

Fig. 19: Modeling Language Definition at the M2 Level

tected resources be identified during requirements analysis and incorporated into different models and how can security requirements be refined during the different analysis and design phases. An initial proposal for integrating Model Driven Security into requirements analysis has been made in [Lodderstedt 2003]. However, more experience carrying out large case studies is needed to answer this question.

APPENDIX

We use MOF as a basis for building and combining modeling languages. Here we provide a more detailed description of this standard. In the context of metamodeling, the term *model* denotes an expression in a modeling language, and we speak of a *metamodel* in case the model itself defines a modeling language. Also, we call the modeling language, in which the metamodel is expressed, a *metamodeling language*. The *MOF metadata architecture* [Object Management Group 2002a] is a reference model for metamodeling. It is comprised of the four meta-levels M0–M3, described in Table VI, which are typically found in a MOF-based environment. The objects in the levels M0–M2 are instances of the objects one level above. So the model at the higher level is the metamodel of the model at the lower level and thus stipulates the vocabulary used to formalize the lower-level model. The MOF model in level M3 is self-describing, i.e., it is modeled using its own vocabulary.

The *MOF Model* at level M3 defines the metamodeling language that is used to define the syntax of modeling languages at the M2 level. The main concepts of the language are class, association, inheritance, and package. We explain below the different language elements, using Figures 19 and 20 as illustration.

A *MOF Class* defines a metaobject at the M2 level, i.e. there are instances of this class at the M1 level with a state and a behavior. In the context of modeling language definition, this means that a MOF class specifies a class of model elements that can be instantiated in a system model.

Figure 19 is specified using the UML profile for MOF as defined in [Object Management Group 2002b]. It depicts the definition of the syntax of a simple system modeling language at the M2 level. The example illustrates how the two MOF

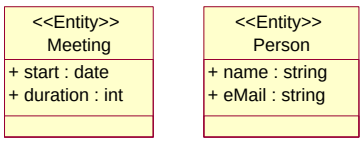

Fig. 20: System Model at the M1 Level

classes Entity and Attribute are specified using UML classes. Figure 20 shows a corresponding system model at the M1 level, given in a UML-based notation. There we see the two entities Meeting and Person, denoted by UML classes with the stereotype «Entity». These entities are instances of the MOF class Entity. Every attribute owned by such a class is automatically considered as an attribute of the entity (MOF class Attribute), so no further stereotypes are necessary.

A *MOF Association* formalizes a relation between two MOF classes. In our example in Figure 19, the classes Entity and Attribute are related by the association EntityAttributes. As a result, the instances of Entity in the system model at the M1 level may have zero or more attributes, as is shown in Figure 20.

A *MOF Attribute* defines a named value holder in each instance of its MOF class. In our example in Figure 19, the attribute name of the MOF class Entity specifies that each entity can have a name. This corresponds to the names "Meeting" and "Person" of the entities in Figure 20.

Each *MOF Operation* specifies a service of its MOF class. These services can be performed on each instance of the MOF class at the model level M1. In our example, there is an operation getAttributeByName that retrieves the attribute of its entity with the given name. MOF operations typically are utility functions used by model transformation functions or modeling tools. Thus, such operations are not visible in system models at the M1 level.

A *MOF Generalization* (not shown in the example) can be used to specify an inheritance relation between MOF classes. If two MOF classes are in a generalization relation then the sub-class derives all attributes, association ends, and operations applicable to its superclass.

A *MOF Package* is the MOF Model construct for grouping elements in a metamodel and it also provides a namespace for naming metamodel elements. Figure 19 shows the package ExampleLanguage, which is the container for the metamodel classes Entity and Attribute

REFERENCES

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

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

Ahn, G.-J. and Shin, M. E. 2000. UML-based representation of role-based access control. In *9th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*. IEEE Computer Society, 195–200.

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

Akehurst, D. and Kent, S. 2002. A relational approach to defining transformations in a metamodel. In *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*. LNCS, vol. 2460. Springer, 243–258.

Beckert, B., Keller, U., and Schmitt, P. H. 2002. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings of the Second Verification Workshop: VERIFY'02* (Copenhagen, Denmark, July 25–26, 2002), S. Autexier and H. Mantel, Eds. DIKU technical reports, vol. 02-07. 113–123.

Bell, D. E. and LaPadula, L. J. 1976. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, The Mitre Corporation. March.

Beyer, D. 2001. *C# COM+ Programming*, Book and CD-ROM (October 15, 2001) ed. John Wiley & Sons.

Brewer, D. and Nash, M. 1989. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 206–214.

Chen, F. and Sandhu, R. S. 1996. Constraints for role-based access control. In *Proceedings of the 1st ACM Workshop on Role-based Access Control*. ACM Press, 39–46.

Damianou, N. 2002. A policy framework for management of distributed systems. Ph.D. thesis, Imperial College, University of London.

Damianou, N., Dulay, N., Lupu, E., and Sloman, M. 2001. The ponder policy specification language. In *Policies for Distributed Systems and Networks (POLICY 2001)*, M. Sloman, J. Lobo, and E. C. Lupu, Eds. Number 1995 in LNCS. Springer-Verlag, 18–38.

Epstein, P. and Sandhu, R. S. 1999. Towards a UML based approach to role engineering. In *Proceedings of the 4th ACM Workshop on Role-based Access Control*. ACM Press, 135–143.

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

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

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

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

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

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

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

Jürjens, J. 2002. UMLsec: Extending UML for secure systems development. In *UML 2002 - The Unified Modeling Language*, J.-M. Jézéquel, H. Hussmann, and S. Cook, Eds. LNCS, vol. 2460. Springer-Verlag, 412–425.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Vol. 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 220–242.

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

Lodderstedt, T. 2003. Model driven security: from uml models to access control architectures. Ph.D. thesis, University of Freiburg, Germany.

Mayfield, T., Roskos, J. E., Welke, S. R., and Boone, J. M. 1991. Integrity in automated information systems. Tech. Rep. 79-91, National Computer Security Center. September.

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

Object Management Group 2002a. *Meta-Object Facility (MOF^{TM}), version 1.4*. Object Management Group. `http://www.omg.org/technology/documents/formal/mof.htm`.

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

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

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

Received Month Year; revised Month Year; accepted Month Year