A Model Transformation Semantics and Analysis Methodology for SecureUML

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff

Information Security, ETH Zurich, 8092 Zurich, Switzerland,
{brucker,doserj,bwolff}@inf.ethz.ch

Abstract SecureUML is a security modeling language for formalizing access control requirements in a declarative way. It is equipped with a UML notation in terms of a UML profile, and can be combined with arbitrary design modeling languages. We present a semantics for SecureUML in terms of a model transformation to standard UML/OCL. The transformation scheme is used as part of an implementation of a tool chain ranging from front-end visual modeling tools over code-generators to the interactive theorem proving environment HOL-OCL. The methodological consequences for an analysis of the generated OCL formulae are discussed.

1 Introduction

Security is a major concern in the development, implementation and maintenance of many distributed software systems like Web services, component-based systems, or database systems. In traditional software engineering practice, the development of a *design model* (business logic) and of a *security model* are treated as completely different tasks; as a consequence, security features are built into an existing system often in an ad-hoc manner during the system administration phase. While the underlying motivation of this practice, a desire for a separation of concerns, is understandable, the conflict between *security requirements* and *availability of services* cannot be systematically analyzed and reasonably balanced in this approach.

An integration of these two aspects into one unified methodology is necessary, ranging from the modeling over the implementation to the deployment and the maintenance phase of a system. To meet this challenge, in [1], a model driven approach has been suggested, which is built upon the SecureUML language. SecureUML is an embedding of a security language for access control into UML class diagrams and statecharts. SecureUML allows for specifying system models and security models within the same visual modeling tool. Subsequent model transformations translate a combined *secured system model* (enriched by a business model implementation) into code including a security infrastructure, e.g., a configuration of policy enforcement points or other access control mechanisms.

While in previous work [1], the semantics of SecureUML has been given in mathematical paper-and-pencil notation for logic and set theory, in this paper,

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 306–320, 2006.
© 2006 Springer-Verlag. This is the author's version of the work. It is posted at http://www.brucker.ch/bibliography/abstract/brucker.ea-transformation-2006 by permission of Springer-Verlag for your personal use. The definitive version was published with doi: 10.1007/11880240_22.

we present its semantics as a model transformation into a secured system model described in plain UML/OCL. In this approach, we take the semantic features of the OCL logic into account (such as undefinedness and three-valuedness), both on the side of the design as well as the security model. Besides the advantage of a seamless integration of SecureUML into the semantic foundations of UML, the approach is the basis of an *implementation* for a tool-chain for SecureUML ranging from visual modeling tools such as ArgoUML to both code-generators and analysis tools such as the proof environment HOL-OCL [4].

The goal of SecureUML is to provide means for a fine-grained specification of access-control requirements like "principals of role r may never access an object of class A" or "method m may never be called on an object of class A satisfying condition c." These properties are essentially temporal safety properties, in the sense that "never something bad will happen." By stating them as requirements, and enforcing them by suitably configured access-control points in an implementation, they may obviously conflict with liveness properties such as "eventually the user will get a result, provided he has permission to it." We show several proof-obligations that are generated for the secured system model to check if it satisfies such desirable properties. These proof-obligations can then be transferred to HOL-OCL and verified by tactic scripts.

Related Work. With UMLsec [6] we share the conviction that security models should be integrated into the software engineering development process by using UML. However, UMLsec provides a formal semantics, but does not provide any tool support, neither for code-generation nor for (formal) model analysis.

M. Koch and F. Parisi-Presicce [7] presented an approach for specifying and analyzing access control policies in UML diagrams. They define an access control semantics using graph transformations into attributed graphs. However, their analysis methodology only considers conflicts, safety, etc. of the security policy itself. In contrast, one of our main contributions is the possibility to reason about the relationship between the security model and the design model.

The Plan of the Paper. After a general introduction into the technical and theoretical foundations, we present the three main contributions: In Section 3, we describe the translation of SecureUML models into standard UML/OCL models (thus providing a translation semantics for the security aspects of a system), in Section 4 we present details over the system architecture and our implementation in a tool chain, and in Section 5, we present several relevant proof obligations representing desirable properties for the secured system model.

2 Technical Background

2.1 SecureUML

SecureUML is a security modeling language based on RBAC [5, 12] with some generalizations. The abstract syntax of SecureUML is defined by the metamodel shown in Figure 1. In particular, SecureUML supports notions of users, roles and

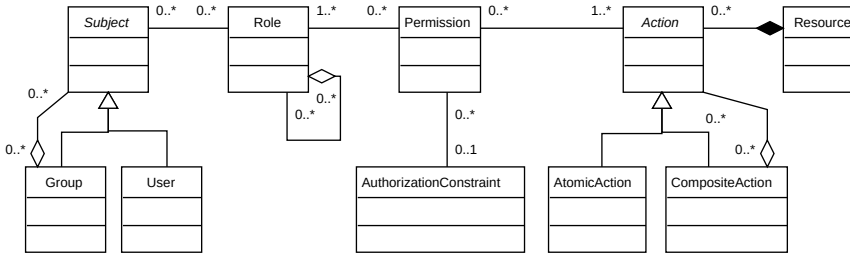

Figure 1. SecureUML Metamodel

permissions, as well as assignments between them: Users can be assigned to roles, and roles are assigned to specific permission. Users acquire permissions through the roles they are assigned to. Moreover, users are organized into a hierarchy of groups, and roles are organized into a role hierarchy. In addition to this RBAC model, permissions can be restricted by *Authorization Constraints*, which are conditions that have to be true (at run-time) to allow access.

Permissions specify which *Role* may perform which *Action* on which *Resource*. SecureUML is generic in that it does not specify the type of actions and resources itself. Instead, these are assumed to be defined in the *design modeling language* which is then "plugged" into SecureUML by specifying (in a SecureUML *dialect*) exactly which elements of the design modeling language are protected resources and what actions are available on them. A dialect may also specify a hierarchy on these actions, so that more abstract actions, like reading a class, can be expressed in terms of lower-level actions, like reading an attribute of the class or executing a side-effect free method. Furthermore, a dialect specifies a *default policy*, i.e., whether access for a particular action is allowed or denied in the case that *no* permission is specified. Usually, and so do we in this paper, one specifies a default policy of *allow* to simplify the security specification.

In previous work, we have presented two dialects: One for a component-based design modeling language, and one for a state-machine based modeling language. Due to limitations of space, we will not address the issue of dialect definitions much further in this paper, and refer to [1] for more details. Instead we will assume as given, without presenting it in detail, a SecureUML dialect definition for UML class diagrams in the spirit of the ComponentUML dialect. This means that the dialect specifies classes, attributes and operations to be resources. The dialect also specifies, among others, the actions *create*, *read*, *update*, and *delete* on classes, *read* and *update* on attributes, and *execute* on operations.

SecureUML features a notation that is based on UML class diagrams, using a UML profile consisting of custom stereotypes. Users, Groups and Roles are represented by classes with stereotypes «secureuml.user», «secureuml.group», and «secureuml.role». Assignments between them are represented by ordinary UML associations, whereas the role hierarchy is represented by a generalization relationship. Permissions are represented as association classes with stereotype

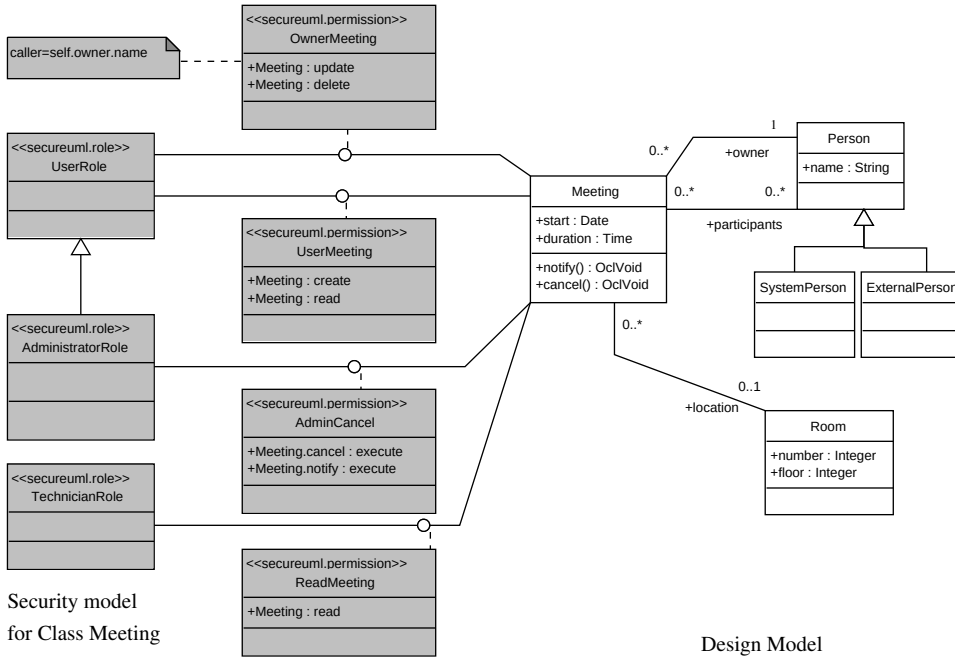

Figure 2. Access Control Policy for Class Meeting

«secureuml.permission» connecting the role and a *permission anchor*. The attributes of the association class specify which action (the attribute's type) on which resource (the attribute's name) is permitted by this permission. Authorization constraints are (OCL) constraints attached to the association class. Note that attributes or operations on roles as well as operations on permission have no semantics in SecureUML and are therefore not allowed in the UML notation.

Figure 2 and 3 show a UML model of a simplified group calendar application together with an exemplary access control policy, which we will use as a running example in this paper.

The left part of Figure 2 shows the access control policy for the class Meeting, whereas the right part shows the design model of the application. The design model consists of *Meetings*, *Rooms*, and *Persons*. Meetings have an owner, participants, and may take place in a particular room. The three association classes specify (from top to bottom) the following access control policy:

1. owners of meetings may delete them, or change the meeting data,
2. ordinary users may read meeting data and create new meetings,
3. administrators may cancel meetings (involves notifying its participants), and
4. technicians may only read meeting data.

For example, the topmost association class (*OwnerMeeting*) has two attributes with type *update* resp. *delete*. This specifies that the associated role (*UserRole*) has the permission to update and to delete meeting objects. According to the policy, however, only *owners* of meetings should be able to do so.

The property of being an owner of a meeting cannot be easily specified using a pure RBAC model. It is therefore specified using the authorization constraint caller = self.owner.name. For this purpose, we introduced a new keyword caller of type String into the OCL language that refers to the name of the authenticated user making the current call. Attaching this authorization constraints to the permission thus restricts the permission to system states where the name of the owner of the meeting matches the name of the user making the request.

The name of the attribute of the association class is used to navigate from the permission anchor, i.e., the classifier associated to the association class, to the actual protected resource. This is necessary because we can only associate classifiers in UML, not operations or attributes. E.g., the permission *AdminCancel* in Figure 2 refers to the operations cancel() and notify() of the class Meeting.

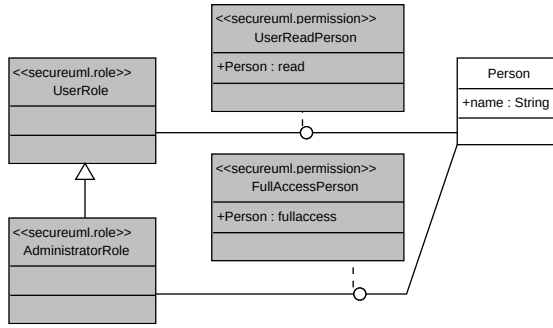

Figure 3. Access Control Policy for Class Person

In addition to Figure 2, Figure 3 specifies the following access control policy for the class Person: 1. ordinary users may read person data 2. administrators have arbitrary access on person.

Note that technicians have no permissions on person objects. Also note that we left out the specification of users, groups and their role assignments in this example to simplify the presentation.

2.2 HOL-OCL

HOL-OCL [4] is an interactive proof environment for UML/OCL. It defines a machine-checked *formalization of the semantics* as described in the standard for OCL 2.0. This is implemented as a conservative, shallow embedding consisting of OCL into the HOL instance of the interactive theorem prover Isabelle [10]. This includes typed, extensible UML data models supporting inheritance and subtyping inside the typed λ-calculus with parametric polymorphism. As a consequence of conservativity wrt. HOL, we can guarantee the consistency of the semantic model. Moreover, HOL-OCL provides several derived calculi for UML/OCL that allows for formal derivations establishing the validity of UML/OCL formulae. Automated support for such proofs is also provided.

3 Transformation

The transformation is based on the idea of substituting the security model, which is specified with SecureUML, with a model of an explicit enforcement mechanism, which is specified in pure UML/OCL. This enforcement mechanism consists of a constant part, i.e., this part is independent of the design model, and a part that varies with the design model. We call the constant part "authorization environment" and explain it in more detail in Section 3.1.

The basic idea of this enforcement mechanism is to model every action on a protected resource by a UML operation and to transform the access control policy into OCL constraints on these operations. Because there are actions on resources that are not operations in the original design model, for example reading or updating an attribute value, we have to transform the design model accordingly. This design model transformation is described in Section 3.2.

Section 3.3 describes the security model transformation, i.e., how the access control policy specified using SecureUML is transformed into OCL constraints.

3.1 Authorization Environment

The basis for our model transformation is a model of a basic authorization environment, as shown in Figure 4.

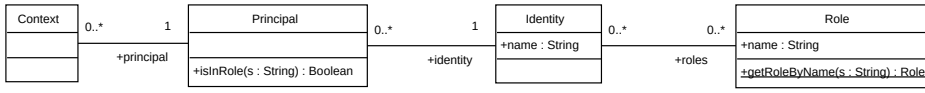

Figure 4. Basic Authorization Environment

All protected resources get a reference to a Context object, which in turn has a reference to a Principal object. Principal objects represent the authenticated users of the system, i.e., the information of the system user together with authentication information. They are associated with their corresponding identity object, which represent the actual system users. To check role membership and user identities, the principal contains an operation isInRole(s: String): Boolean. The class Identity holds information about the system user(s), which in our case is just its name and its roles. The distinction between Principal and Identity allows a certain flexibility in the treating of authenticated users. For example, they can hold information about the authentication method they used. Also, it allows users to authenticate for a session using only a subset of their assigned roles (which is currently not supported in SecureUML). In the simplified model presented here, the principal object does not hold any extra information, and system users will always have all their assigned roles. This is done by imposing the following constraint:

context Principal :: isInRole(s: String) : Boolean
post: result = self.identity.roles.name->includes(s)

This environment is minimal on purpose, but sufficient to express authorization requirements. In particular, we do not consider authentication here.

3.2 Design Model Transformation

The model transformation is split into two parts: transforming the design model, and transforming the security model. Transforming the design model is necessary to allow the expression of security policies as OCL constraints. The transformation itself consists of first copying the input design model, adding the authorization environment to it, and adding new (access controlled) operations to the model. In particular, all invariants, preconditions and postconditions of the original design model are preserved, and new constraints are only imposed on generated classes and operations.

For the addition of the authorization environment, we associate each permission anchor with the context class from the authorization environment. Furthermore, all access controlled actions have to be represented as operations in the target model. Table 1 gives an overview over the operations that are generated in this step, and how their semantics is specified using OCL postconditions.

model element	generated operation with OCL constraints
Class C	**context** C::new():C **post**: result.oclIsNew() and result->modifiedOnly()
	context C::delete():OclVoid **post**: self.oclIsUndefined() and self@pre->modifiedOnly()[a]
Attribute att	**context** C::getAtt():D **post**: result=self.att
	context C::setAtt(arg:D):OclVoid **post**: self.att=arg and self.att->modifiedOnly()
Operation op	**context** C::op_sec (...):... **pre**: pre_{op} **post**: $\overline{\mathrm{post}}_{op} = \mathrm{post}_{op}[\mathsf{f}() \mapsto \mathsf{f\_sec}(), \mathsf{att} \mapsto \mathsf{getAtt}()]$

[a] While self@pre is unsupported by the concrete syntax, it is semantically well-defined.

Table 1. Overview of generated operations

For example, reading and writing an attribute value has to be represented by getter- and setter-methods. This means that for each attribute with public visibility, a public getter and a public setter method has to be generated, and the visibility of the attribute has to be made private. This transformation is similar in spirit to what one has to do when generating executable code or code skeletons from the model, cf. [1] for example. Instead of generating code for these getter and setter methods, we here have to generate OCL constraints to define their semantics. As a consequence, we generate the postconditions shown in Table 1.

Also, for each operation op() in the design model, we generate a second operation op_sec(). The postcondition $\overline{\mathrm{post}}_{op}$ for op_sec() is structurally the same as the postcondition post_{op} for op(), where every occurrence of an attribute call is

substituted with the corresponding getter operation call, and every occurrence of an operation call is substituted with the corresponding call of the secured operation. This substitution ensures that the *functional* behavior of the secured operation stays the same, but that it is only "executable" when all security requirements for establishing the postcondition are fulfilled. The reasoning here is that a caller will need (at least) the permission necessary to establish the postcondition for performing an operation call. Furthermore, we make the precondition pre_{op} specified for op() into a precondition for op_sec(), too. For this, we keep the OCL expression unchanged, i.e., no substitutions are necessary this time, and only change the context declaration of the OCL constraint.

In the postcondition of setter methods, i.e., C::setAtt(arg:D):OclVoid, it is not sufficient to specify that the attribute gets the value of the given argument. We also need to specify that "nothing else" happens during this operation call. Using standard OCL this is difficult or even impossible for arbitrary methods: one has to specify that the whole system stays unchanged *except* for this attribute. Therefore, HOL-OCL provides an extension of OCL for specifying frame properties within postconditions: Set(T)::modifiedOnly():Boolean. This allows for specifying explicitly the set of object instances that the system can change during state transition. For example, we can now define C::setAtt(arg:D):OclVoid using the postcondition self.att = arg and self.att->modifiedOnly().

Analogous transformations are done for association ends, i.e., they are handled as they were attributes. Also, operations for constructing and deleting objects are created, with the given constraints specifying their semantics.

Figure 5 shows the generated authorization environment together with the transformed permission anchors of the running example.

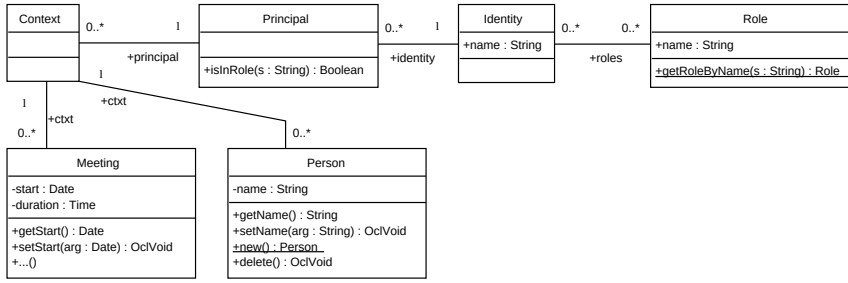

Figure 5. Authorization Environment with Permission Anchors

3.3 Security Model Transformation

Role Hierarchy First, we transform the role hierarchy of the security model into OCL invariant constraints on the classes of the authorization environment. The total set of roles in the system is specified by enumerating them:

context Role **inv**: Role.allInstances().name=Bag{<List of Role Names>}

The inheritance relation between roles is then specified by an OCL invariant constraint on the Identity class:

```
context Identity inv: self . roles .name−>includes('<Role1>') implies
                      self . roles .name−>includes('<Role2>')
```

Given this, role assignments to identities can then be stated by further OCL invariant constraints on the Identity class:

```
context Identity inv: self .name = '<userName>' implies
                      self . roles .name=Bag{<List of Role Names>}
```

We denote by inv_{sec} the conjunction of these invariants. We have to ensure that inv_{sec} is consistent, i.e., that situations like the following do not arise:

```
context Role
inv: Role. allInstances (). name=Bag{'UserRole','AdministratorRole','TechnicianRole'}

context Identity inv: self .name = 'Alice' implies self . roles .name=Bag{'Spy'}
```

Security Constraints The main part, however, of the security model transformation is the generation of the security constraints for the operations generated during the design model transformation. The existing constraints on the generated operations are transformed according to Table 2.

Effect of the Security Model Transformation

inv_C	$\mapsto \text{inv}_C$
pre_{op}	$\mapsto \text{pre}_{op}$
post_{op}	$\mapsto$ let auth = auth_{op} in if auth then $\overline{\text{post}}_{op}$ else result . oclIsUndefined () and Set{}−>modifiedOnly() endif

Table 2. Overview of Transformed Constraints

Table 2 applies only to operations generated during the design model transformation. As noted above, the pre-existing model elements of the design model are preserved. Only the postconditions are changed during this transformation, i.e., the invariants inv_C for classes C of the design model and the preconditions pre_{op} for access-controlled operations stay the same. The transformation wraps the postcondition generated during the design model transformation with an access control check using the authorization expression auth_{op}, which evaluates to true if access is granted, and false otherwise. If access is granted, the behavior of this operation will not be changed. Otherwise, the transformed postcondition ensures that no result is returned and the system state does not change.

The expression auth_{op} is built in the following way: Let $\text{perm}_1, ..., \text{perm}_n$ be the permissions for this operation call, and let roles_i be the set of roles, constr_i be the authorization constraint associated with permission perm_i, and

$\overline{\text{constr}_i} = (\text{constr}_i[\text{caller}\mapsto\text{ctxt.principal.identity.name}])$
$[\text{f}() \mapsto \text{f@pre}(), \text{att} \mapsto \text{att@pre}, \text{aend} \mapsto \text{aend@pre}]$

be the OCL expression where every occurrence of the non-standard keyword caller in constr_i is substituted by the expression ctxt . principal . identity .name, which evaluates the name of the current caller using the authorization environment. Operation, attribute, and association end calls are substituted by their post-state equivalents. auth_{op} is then defined as the following OCL expression:

```
auth_op := let perm1:Boolean = Set{<list of role names r ∈ roles_1>}
                               ->exists(s|ctxt@pre. principal@pre . isInRole@pre(s))
                               and constr_1
           -- analogous for perm2 to permn
           perm:Boolean = perm1 or perm2 or ... permn
         in perm.oclIsDefined() and perm
```

We explicitly check the authorization expression for undefinedness, mapping it to false if it is undefined. This is necessary because undefinedness can be caused by user-specified authorization constraints, which form a part of auth_{op}.

For illustration purpose, we show the final postcondition of the setter operation Meeting:: setStart () below:

```
context Meeting:: setStart (arg:Date):OclVoid
post: let auth = let perm1:Boolean = Set{'UserRole'}
                                 ->exists(s|ctxt@pre. principal@pre . isInRole@pre(s))
                                 and ctxt@pre. principal@pre . identity@pre .name@pre
                                     = self.owner@pre.name@pre
                 perm2:Boolean = Set{'AdministratorRole'}
                                 ->exists(s|ctxt@pre. principal@pre . isInRole@pre(s))
       in perm_1 or perm_2
   in if auth. oclIsDefined () and auth then true
      else result . oclIsUndefined () and Set{}->modifiedOnly() endif
```

4 Implementation

The transformation is part of a tool-chain (see Figure 6) that consists of a UML CASE tool with an OCL type-checker for modeling software systems, a model repository, model analyzers and various code generators.

We use the UML CASE tool ArgoUML (`http://argouml.tigris.org`) and combine it with the Dresden OCL2 Toolkit (`http://dresden-ocl.sf.net/`), which provides a OCL 2.0 compliant [11] parser and type-checker. Both tools use the Netbeans Metadata Repository (MDR), which is a model repository supporting the OMG MOF and the Java JMI standards. Using MDR, one can instantiate arbitrary MOF-compliant metamodels, which results in a *model extent*, a container for models compliant with this metamodel. MDR can automatically generate JMI interfaces from the metamodel so that one can, using these interfaces, access and manipulate the contents of such a model extent.

Our Java-based transformation tool, *su2holocl*, uses different MDR extents, namely: As first step of the transformation we parse the input model using the SecureUML *profile*, into a separate model extent based on the SecureUML *metamodel*. This gives us the ability to deal with the security part of the model on an

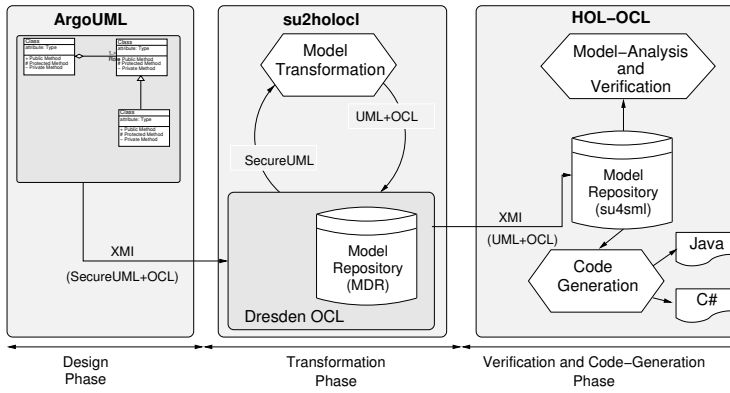

Figure 6. Tool-chain Overview

abstract level. The Dresden OCL Toolkit uses a specialized metamodel combining the UML 1.5 and the OCL 2.0 metamodel. This results in an upward compatible extension of the UML 1.5 Metamodel: every UML 1.5 model is still a model of the combined metamodel. We use the OCL type-checker for checking user-defined constraints that occur in the design-model and for checking the security constraints that are generated during the transformation. As we currently only typecheck the transformed OCL constraints, not the original authorization constraints, we do not need to extend the Dresden OCL Toolkit, e.g., for supporting "caller" as a new keyword. The toolkit also provides an OCL expression visitor, which we use to implement the substitutions for transformed postconditions.

For interfacing the results of our model transformation with Isabelle and HOL-OCL, which are written in SML, we also developed a data repository: *su4sml*. This repository is also implemented in SML and supports the various metamodels we are using, e.g., UML, OCL, SecureUML. At the moment, su4sml is used for importing UML models into HOL-OCL. We also developed a generic code-generator based on su4sml that generates code from SecureUML models in various SecureUML dialects, that respects the specified access control policy.

5 Methodology

In this section, we discuss three key issues that arise while adding access control specifications to an object-oriented system model. In particular, we define several well-formedness conditions on the security specification: an access control aware variant of Liskov's principle, a data-accessibility condition and a notion of relative consistency.

5.1 Access Control and Inheritance

In an object-oriented system, *inherited methods* inherit the access control policy assigned to the method in the superclass. However, one can assign an access

control policy to a subclass which is completely independent from the inherited policy. This leads to the idea of extending Liskov's principle to access control policies, i.e., access rights should be preserved along the class hierarchy. This boils down to the following two well-formedness conditions:

1. all overridden methods must have less or equal role assignments as their counterparts in superclasses and
2. the security constraints for an overridden operation must imply the corresponding security constraints of the original operation.

A secured system model satisfying these requirements is called *overriding-secure*.

Note that the implication required here goes in the opposite direction of Liskov's principle [8]. We want to rule out security problems caused by overridden methods that have more functionality and therefore need a more restrictive access control policy. The overriding-secure property is therefore advisable, although violations may be adequate in certain situations.

5.2 Accessibility of Data

The following problem comparable to "dead-code-detection" in conventional compilers may occur in a secured system model. It is not necessarily implying inconsistency (see next subsection), but indicating bad specification practice potentially resulting from specification errors.

Using potentially inconsistent security constraints may lead to the situation that some operation in a class can be accessed by no principal. We call an operation of this kind *inaccessible*.

Accessibility of an operation op in class C may be defined as follows:

1. if op has no role assignments, it is accessible by all principals (following our default-accessibility rule (c.f. Section 2.1)).
2. if op has role-assignments labeled with security constraints $SC_1, \ldots, SC_n$, then op is accessible iff $SC_1 \vee \ldots \vee SC_n$ holds for all objects of this class.

As a well-formedness condition of a secured system model, we require that all operations are accessible.

5.3 Relative Consistency

Following general practice, we call a system model *consistent* iff the conjunction of all invariants $\text{inv}_{\text{global}}$ is *invariant-consistent* and all operations m are *implementable*. An invariant inv is *invariant-consistent* iff there are satisfying states (i.e., $\exists \sigma.\sigma \models inv$ in the terminology of [11, Appendix A]). An operation m is *implementable* iff for all pre-states σ_{pre} and all input parameter *self*, i_1, ..., i_n there exist a post-state σ_{post} and an output *result* such that the operation specification of m (consisting of pre_{op} and post_{op}) can be satisfied:[1]

[1] We make the implicit binding of the internal free variables $self, i_1, \ldots, i_n$ occurring in the OCL formulae pre_{op} and post_{op} explicit.

$$\forall\, \sigma_{\text{pre}} \in \Sigma, \mathit{self}, i_1, \ldots, i_n.\ \sigma_{\text{pre}} \models \text{pre}_{op}(\mathit{self}, i_1, \ldots, i_n) \longrightarrow$$
$$\exists\, \sigma_{\text{post}} \in \Sigma, \mathit{result}.\ (\sigma_{\text{pre}}, \sigma_{\text{post}}) \models \text{post}_{op}(\mathit{self}, i_1, \ldots, i_n, \mathit{result})$$

where Σ is the set of legal states (i.e., $\Sigma = \{\sigma | \sigma \models \text{inv}_{\text{global}}\}$). Our notion of implementability of an operation is only meaningful for system models where $\text{inv}_{\text{global}}$ is invariant-consistent; otherwise the above definition yields true for the trivial reason that Σ is empty. Being implementable is also called "non-blocking" in the literature and can be viewed as a liveness property.

The question arises what is the "desirable semantic result" of our model transformation on the design model. In particular, we expect that in case of a security violation (i.e., auth_{op} does not hold) an operation preserves the state and reports an error. In the other case (i.e., auth_{op} does hold, meaning that a principal has "enough" permissions), we expect that the model transformation preserves the "functional content" of the operation specification of the system model. These requirements are captured by a *security proof obligation* spo_{op} (which is automatically generated for each operation):

$$\text{spo}_{op} := \text{auth}_{op} \ \ \mathsf{implies}\ \text{post}_{op} \triangleq \overline{\text{post}}_{op}$$

where $x \triangleq y$ is the strong equality yielding true iff $x = y$ (i.e., the strict OCL equality holds) or x and y are both undefined.

The following example illustrates the role of security proof obligations, and what sorts of inconsistencies in secured system models they rule out. Assume that we want to add to the class Meeting the operations:

```
context Meeting::getNames(): Sequence(String)
post:    result = self.participants.name->asSequence()

context Meeting::getSize(): Integer
post:    result = self.participants->size()
```

and attempt to give execute permissions for both operations to TechnicianRole. Recall that this role has no read permissions for objects of class Person and therefore is not able to access the names of participants. Following the definitions in Section 3, we have:

$$\overline{\text{post}}_{\text{getNames}} \triangleq \mathsf{result} = \mathsf{self}.\,\mathsf{getParticipants}().\,\mathsf{getName}().\mathsf{asSequence}()$$

Since auth_{op} and the strong equality $(\_ \triangleq \_)$ never reduce to OclUndefined, the security proof obligation $\text{spo}_{\text{getNames}}$ boils down to:

$$\sigma \models \text{auth}_{op} \longrightarrow (\sigma, \sigma') \models \text{post}_{\text{getNames}} \triangleq \overline{\text{post}}_{\text{getNames}}$$

However, under the assumption $\sigma \models \text{auth}_{op}$ the caller is in the role TechnicianRole, i.e., has execute permission to Meeting::getNames() in the given concrete state σ. Because users in the role TechnicianRole do not necessarily have permission for the accessor Person::getName(), this operation call may yield undefined. In this case, $\overline{\text{post}}_{\text{getNames}}(\mathit{self}, \mathit{result}) = \mathsf{OclUndefined}$. For consistent design models, however, $\text{post}_{\text{getNames}}(\mathit{self}, \mathit{result})$ is never OclUndefined. Therefore,

the conclusion becomes false and the security proof obligation becomes invalid: $\text{spo}_{\text{getNames}} =$ false . This indicates that it does not make sense to give permissions for the operation Meeting::getNames() to the TechnicianRole role, as they cannot execute it anyways. In contrast, we can prove $\text{spo}_{\text{getSize}}$ because read permission for the association end participants is sufficient to satisfy the postcondition. As the TechnicianRole has this permission, we can grant the TechnicianRole role the execute permission for Meeting::getSize().

Due to the construction of $\overline{\text{post}}_{op}$ and the accessor functions, the proof or disproof of spo_{op} is fairly easy and can be automatically supported in the most common case: a non-recursive postcondition containing just attribute accesses. For recursive calls induction is needed. An important property of security proof obligations is illustrated by the following theorem:

Theorem 1. *An operation op_sec of the secured system model is implementable provided that the corresponding operation of the design model is implementable and* spo_{op} *holds.*

Proof. The complete proof can be found in the extended version of this paper [2].

Inaccessible operations (as discussed in the previous section) were transformed to totally undefined functions. They are clearly implementable operations, albeit pathological ones.

A class system is called *security consistent* if all spo_{op} hold.

Theorem 2. *A secured system model is consistent provided that the design model is consistent, the class system is security consistent, and the security model is consistent.*

Proof. By definition of the model transformation, we have $\text{inv}_{\text{sec-global}} \equiv \text{inv}_{\text{global}}$ and inv_{sec}. Since the invariant of the security model is consistent, since $\text{inv}_{\text{global}}$ is invariant-consistent by assumption, and since the signature parts of the security model and the design model are disjunct, there must be states that satisfy both invariants. The implementability of all methods follows from Theorem 1. □

These theorems enable modular specifications and reasoning for secure systems, which is important for large-scale applications.

6 Conclusions

We presented a systematic approach to include access control into data models given by UML class diagrams. From an integrated design and security model, a secured system model is generated which can be analyzed for consistency and liveness properties on the one hand and further transformed to code on the other.

Access control is a necessary means to establish security, but not a sufficient one: class invariants or implementation details may allow an attacker to infer implicit secrets of a system. For example, the Name attribute in Person may be correlated via class invariants to other attributes that can be accessed by

TechnicianRole. A systematic analysis of this problem on the basis of the secured system model requires data flow analysis (see [9, Sect. 5], for an overview) which is out of the scope of this paper, but clearly an interesting line of future research.

Another line of future research is proving that the generated code—including the code for the methods of the design model—complies to the secured system model, or that a more concrete secured system model represents a *refinement* of a more abstract one. This involves proofs over the correctness of implementation issues of access control points as well as auxiliary data or different data structures which need different internal checks to establish the security behavior specified in the original secured system model. This type of verification problems has already been addressed [3]; however, it remains to show how they can be applied to an object-oriented setting and SecureUML.

References

[1] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. ACM *Trans. Softw. Eng. Methodol.*, 15(1), 2006.
[2] A. D. Brucker, J. Doser, and B. Wolff. A model transformation semantics and analysis methodology for SecureUML. Tech. Rep. 524, ETH Zürich, 2006.
[3] A. D. Brucker and B. Wolff. A verification approach for applied system security. *Int. Journal on Software Tools for Technology*, 7(3):233–247, 2005.
[4] A. D. Brucker and B. Wolff. The HOL-OCL book. Tech. Rep. 525, ETH Zürich, 2006.
[5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. ACM *Trans. Infor. and System Security*, 4(3):224–274, 2001.
[6] J. Jürjens. *Secure Systems Development with* UML . Springer, 2004.
[7] M. Koch and F. Parisi-Presicce. Access control policy specification in UML. In *Critical Systems Development with* UML , pp. 63–78. 2001. TUM-I0208.
[8] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. ACM *Trans. Progr. Lang. and Systems*, 16(6):1811–1841, 1994.
[9] H. Mantel. Information flow control and applications – bridging a gap. In J. N. Olivera and P. Zave, eds., FME , LNCS , vol. 2021, pp. 153–172. Springer, 2001.
[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/*HOL *— A Proof Assistant for Higher-Order Logic*, LNCS , vol. 2283. Springer, 2002.
[11] UML 2.0 OCL specification. 2003. Available as ptc/2003-10-14.
[12] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.