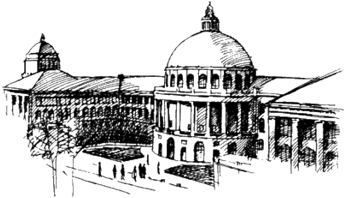
A Model Transformation Semantics and Analysis Methodology for SecureUML

Achim D. Brucker

joint work with

Jürgen Doser, and Burkhart Wolff

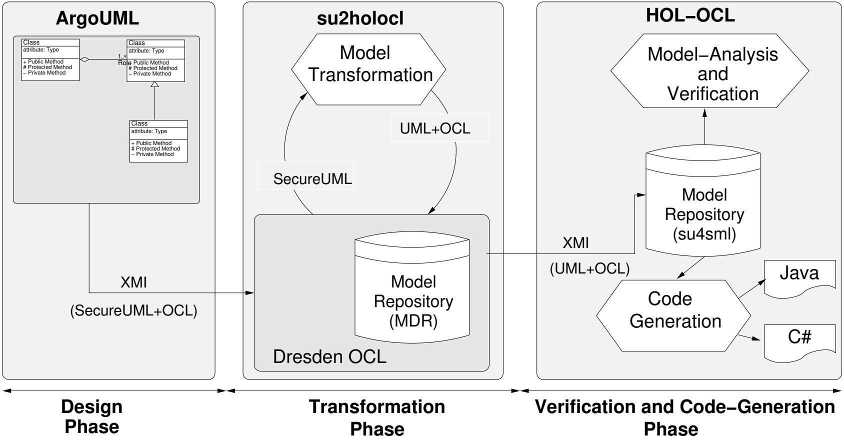


Information Security, ETH Zurich, Switzerland

Model-Driven Engineering Languages and Systems
October 4, 2006

Outline

Our Vision


ArgoUML
Class
attribute: Type
+ Public Method
# Protected Method
– Private Method
1..*
Role
su2holocl
Model Transformation
UML+OCL
SecureUML
Model Repository (MDR)
Dresden OCL
XMI
(SecureUML+OCL)
HOL–OCL
Model–Analysis and Verification
Model Repository (su4sml)
XMI
(UML+OCL)
Code Generation
Java
C#
Design Phase
Transformation Phase
Verification and Code–Generation Phase

Modeling Access Control with SecureUML

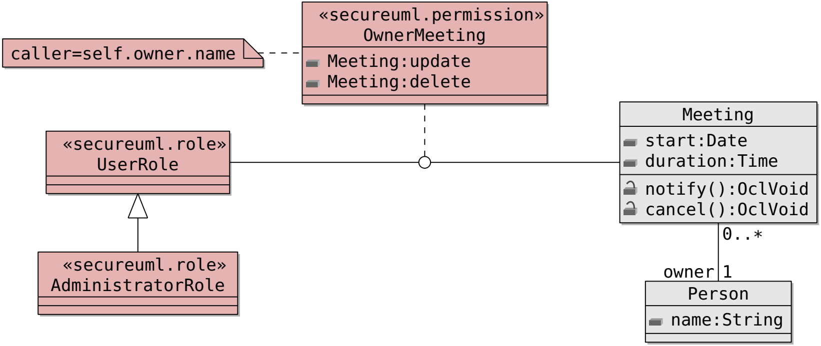
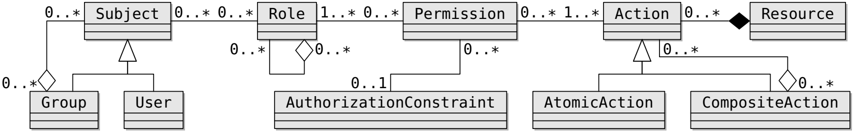


Figure: Access Control Policy for Class Meeting Using SecureUML

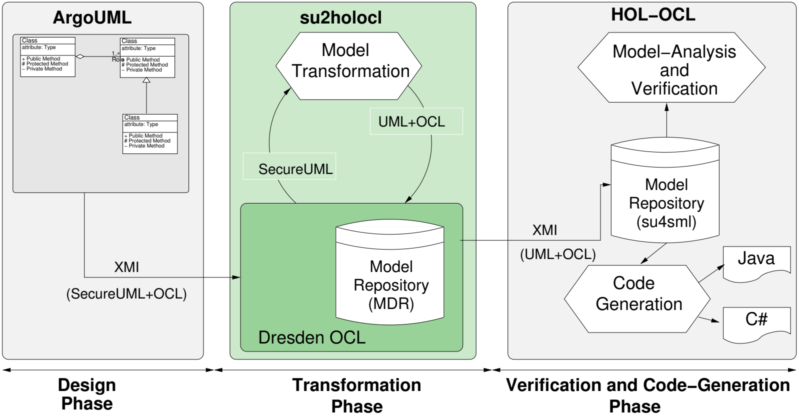
SecureUML



SecureUML

- is a UML-based notation,
- provides abstract Syntax given by MOF compliant metamodel,
- is pluggable into arbitrary design modeling languages,
- is supported by an ArgoUML plugin.

The Model Transformation



From SecureUML to UML/OCL

Substitute the SecureUML model by an explicit enforcement model using UML/OCL.

The transformation basically

1. initializes a concrete authorization environment,
2. transforms the design model,
3. transforms the security model.

The Authorization Environment

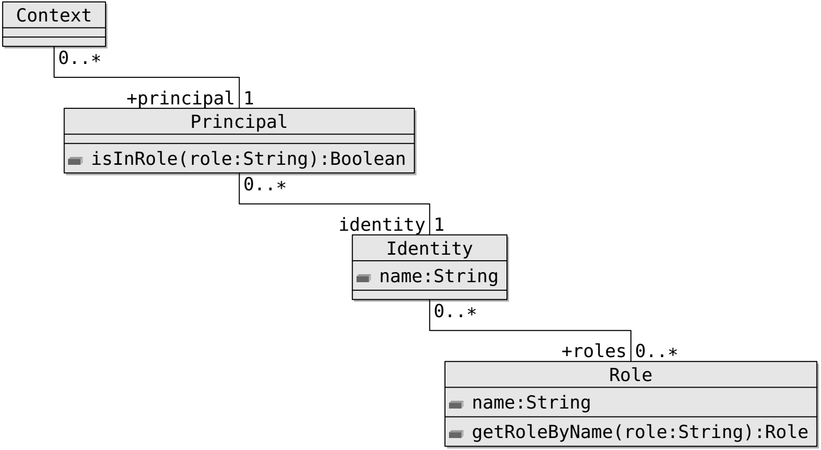


Figure: Basic Authorization Environment

Design Model Transformation

Generate *secured* operations for each class, attribute and operation in the design model.

- for each class `C` we add constructors and destructors,
- for each attribute of class `C` we add getter and setter operations, and
- for each operation `op` of class `C` we add a secured wrapper:

```
context C::op_sec(...):...
  pre: pre_op
  post: \overline{post_op} = post_op[f() ↦ f_sec(), att ↦ getAtt()]
```

Security Model Transformation

- The role hierarchy is transformed into invariants for the `Role` and `Identity` classes,
- Security constraints are transformed as follows:

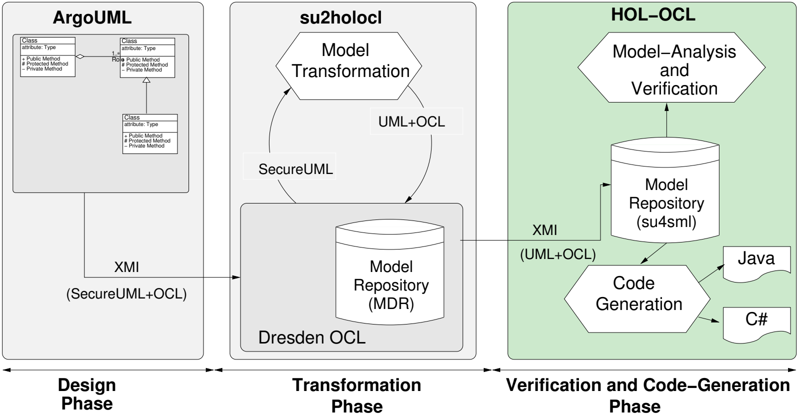
$\text{inv}_C \mapsto \text{inv}_C$

$\text{pre}_{op} \mapsto \text{pre}_{op}$

$\text{post}_{op} \mapsto$ `let auth =` auth_{op} `in`
`if auth`
`then` $\overline{\text{post}}_{op}$
`else result.oclIsUndefined()`
`and Set{}->modifiedOnly()`
`endif`

where auth_{op} represents the authorization requirements.

Consistency Analysis



Relative Consistency

- An invariant is invariant-consistent, if a satisfying state exists:

$$\exists \sigma.\ \sigma \vDash \mathit{inv}$$

- A model is global consistent, if the conjunction of all invariants is invariant-consistent:

$$\exists \sigma.\ \sigma \vDash \mathit{inv}_1 \ \mathsf{and}\ \mathit{inv}_2 \cdots \mathsf{and}\ \mathit{inv}_n$$

- An operation is implementable if for each satisfying pre-state there exists a satisfying post-state:

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

Proof Obligations

- We require:
  - if a security violation occurs, the system state is preserved
  - if access is granted, the model transformation preserves the functional behavior

  Which results for each operation in a *security proof obligation*:

$$\mathrm{spo}_{op} := \mathrm{auth}_{op}\ \texttt{implies}\ \mathrm{post}_{op} \triangleq \overline{\mathrm{post}}_{op}$$

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

Modularity Results

Our method allows for a modular specifications and reasoning for secure systems.

Theorem (Implementability)

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.*

Theorem (Consistency)

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.

Conclusion

We presented

- a modelling approach including access control,
- a toolchain supporting our approach,
- a method for consistency analysis of access control specifications.

Future work includes,

- automatic generation of proof obligations,
- analyzing case studies,
- better proof support for access control specifications.

Appendix

HOL-OCL

HOL-OCL

- provides formal, machine-checked semantics for OCL 2.0,
- servers as a basis for examining extensions of OCL,
- is an interactive theorem prover for OCL (and UML class models),
- publicly available:
  `http://www.brucker.ch/projects/hol-ocl/`.

Demo available!

Design Model Transformation: Classes

- for each class `C`

```
context C::new():C
  post: result.oclIsNew() and result->modifiedOnly()
context C::delete():OclVoid
  post: self.oclIsUndefined() and self@pre->modifiedOnly()
```

Design Model Transformation: Attributes

- for each Attribute `att` of class `C`

```
context C::getAtt():T
  post: result=self.att
context C::setAtt(arg:T):OclVoid
  post: self.att=arg and self.att->modifiedOnly()
```

Design Model Transformation: Operations

- for each Operation `op` of class `C`

```
context C::op_sec(...):...
  pre: pre_op
  post: \overline{post}_op = post_op[f() ↦ f_sec(), att ↦ getAtt()]
```

Security Model Transformation: Role Hierarchy

- 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>')
```