Model Driven Security: Foundations, Tools, and Practice

David Basin, Manuel Clavel, and Marina Egea

ETH Zurich, IMDEA Software Institute

Thursday 1st, 2011

Outline I

Questions that you may want to ask about your security-design models

Questions about your security policy

1. *"per se"*: e.g., are supervisors allowed to delete meetings? under which general conditions (i.e., authorization constraints)?
2. but *applied to a concrete snapshot of the system*: e.g., can Bob delete the kick-off meeting of the NESSoS project?
3. but *applied to scenarios in general*: e.g., can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own?

Questions that you may want to ask about your security-design models applied to scenarios in general

Your question is whether there exists a scenario in which

- the authorization constraints for the given role to perform the given action (e.g., the "caller" being the owner of the meeting to be deleted) hold *AND*
- the invariants of your system hold *AND*
- your *additional contraints* (e.g.., the "caller" not being the owner of the meeting to be deleted) also hold.

And many other questions, of course. But, for the type of the above questions, we have a way of getting the answers.

Method for getting the answers to questions about your security-design models of Type I.

Type I. Questions about your security policy *"per se"*: e.g., are supervisors allowed to delete meetings?

- Step 1 Formalize your question as an OCL query, using the language provided by the SecureUML+ComponentUML metamodel. This step is manual, but you can use our "library" of predefined queries in OCL about security-design models.
- Step 2 Evaluate your OCL query on the instance of the SecureUML+ComponentUML metamodel corresponding to your security-design model. This step is automatic.

Method for getting the answers to questions about your security-design models of Type II.

Type II. Questions about your security policy, but *applied to a concrete scenario*: e.g., can Bob delete the kick-off meeting of the NESSoS project?

Step 1 For each of the roles assigned to the given user in the given scenario, query your security policy about the conditions for users in that role to be allowed to perform the given action. This step is manual, and follows the same metamodel-based technique explained before.

Step 2 Evaluate the conditions obtained on the given scenario. This step is automatic.

Method for getting the answers to questions about your security-design models of Type III.

Type III. Questions about your security policy, but *applied to scenarios in general*: e.g., can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own?

- Step 1 Query your security policy about the conditions for users in the given role to be allowed to perform the given action. This step is manual, and follows the same metamodel-based technique explained before.
- Step 2 Formalize in OCL your additional contraints.
- Step 3 Translate into first-order logic the obtained conditions, your additional constraints, and the invariants of your system. This step is automatic.
- Step 4 Use standard first-order theorem-prover techniques to solve the problem; SMT solvers may give you the solution automatically.

Outline I

The Object Constraint Language (OCL)

- OCL was used to specify constraints on OO systems models.
  - Used to specify class invariants, pre- and post-conditions for methods, as a navigation language, as a query language, etc..
- Textual modeling language.
- Adds precision to imprecise models.
- Used as a query language for model analysis.
- Standardized.
- Supported by IBM, Microsoft, Oracle, etc..

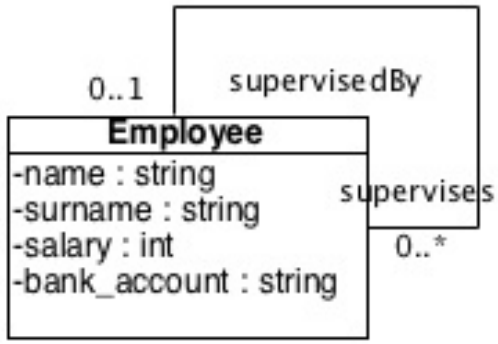
OCL

OCL is a formal specification language that is implementable.

- Supports object concepts and has an intuitive syntax.
- Not a programming language: no control flow, no-side effects.
- Useful at the software design and analysis stages, for the V&V of systems.
- Tool support:
  - evaluators (MDT-OCL, EOS, Borland),
  - compilers (MySQL4OCL),
  - interactive provers (HOL-OCL), SMT solvers (OCLSat4Yices), etc..

OCL Examples

1. Employee.allInstances()->size()
2. context Employee inv: self.supervisor->size() <= 1
3. Employee.allInstances().name->includes('John')
4. Employee.allInstances().name->asSequence()->indexOf('John')
5. Employee().allInstances()->forAll(e| e.salary > 800)



OCL Type system

OCL is a strongly typed language.

- Basic types: String, Integer, Boolean, and Real.
- Class types: Employee etc..
- Collection types:
  Bag(Integer) = Bag{1,2,4,2,3,4,4},
  Bag(Employee) = Bag{e1,e2,e2,e4},
  Sequence(Integer) = Sequence{1,2,4,2,3,4,4}.

Types of OCL operations

- Operations on basic types: concat, substring, not, implies, +, -, abs(), floor(), ...
- Class operations: Employee.allInstances()
- Collection operations:
  Bag{1,2,4,2,3,4,4}->count(2),
  Sequence{1,2,4,2,3,4,4}->at(3)
- Iterator operations:
  Employee.allInstances()->select(e| e.salary > 1000),

  Employee.allInstances()->forall(e, e' | e<>e' implies e.bankaccount <> e'.bankaccount)

Frequently used collection operations

- Collection operations: <src-exp>->nameOp(<exp>)
  - boolean operations: excludesAll/includesAll, isEmpty/notEmpty, excludes/includes, =, <>, ...
  - numeric operations: count, size, sum, ...
  - operations over collections: including, excluding, union, ...
- Iterator operations: <src-exp>->nameIterOp(v | <body>)
  - boolean body: any, exists, forAll, reject/select
  - any type body: collect

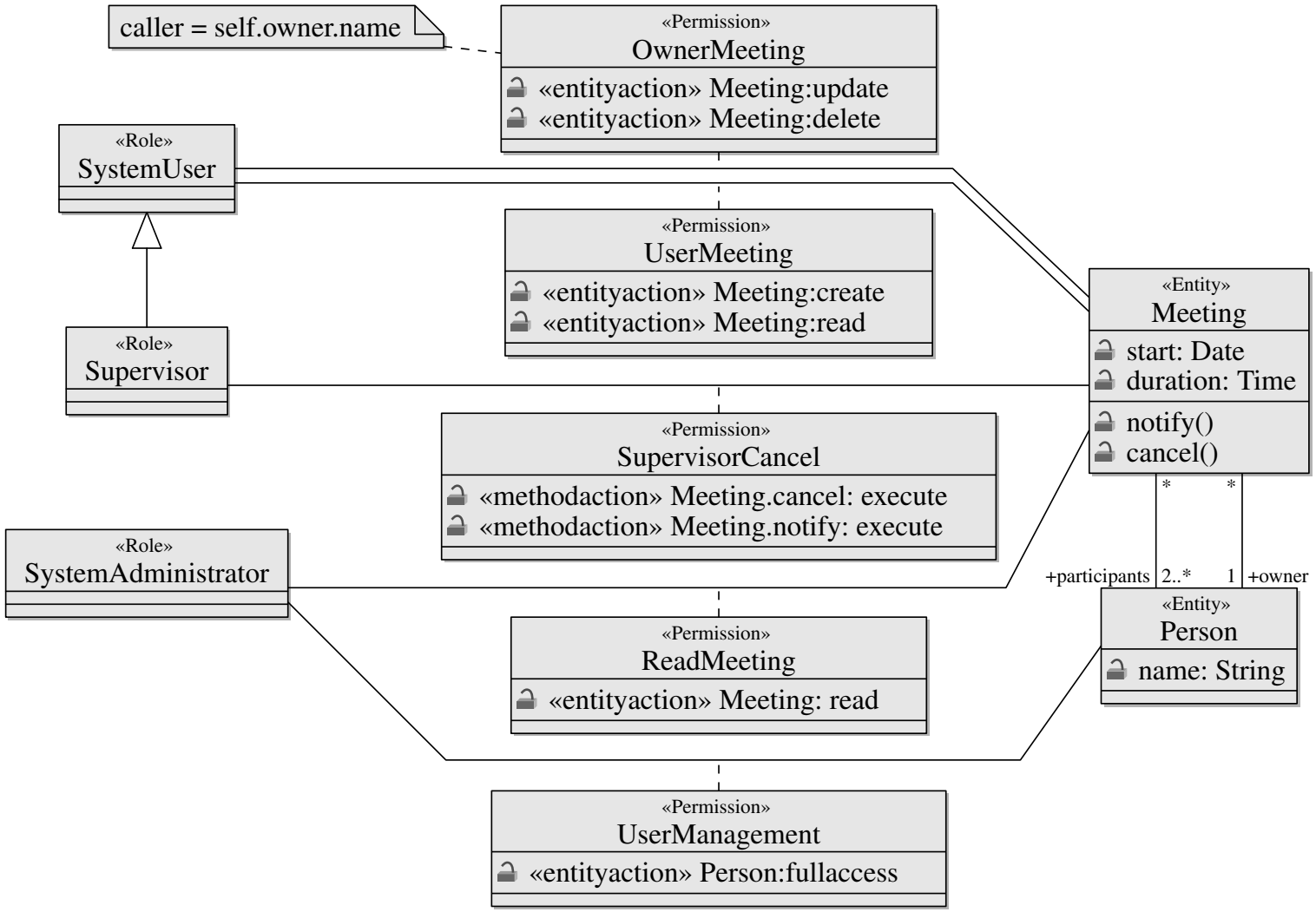
Outline I

Outline I

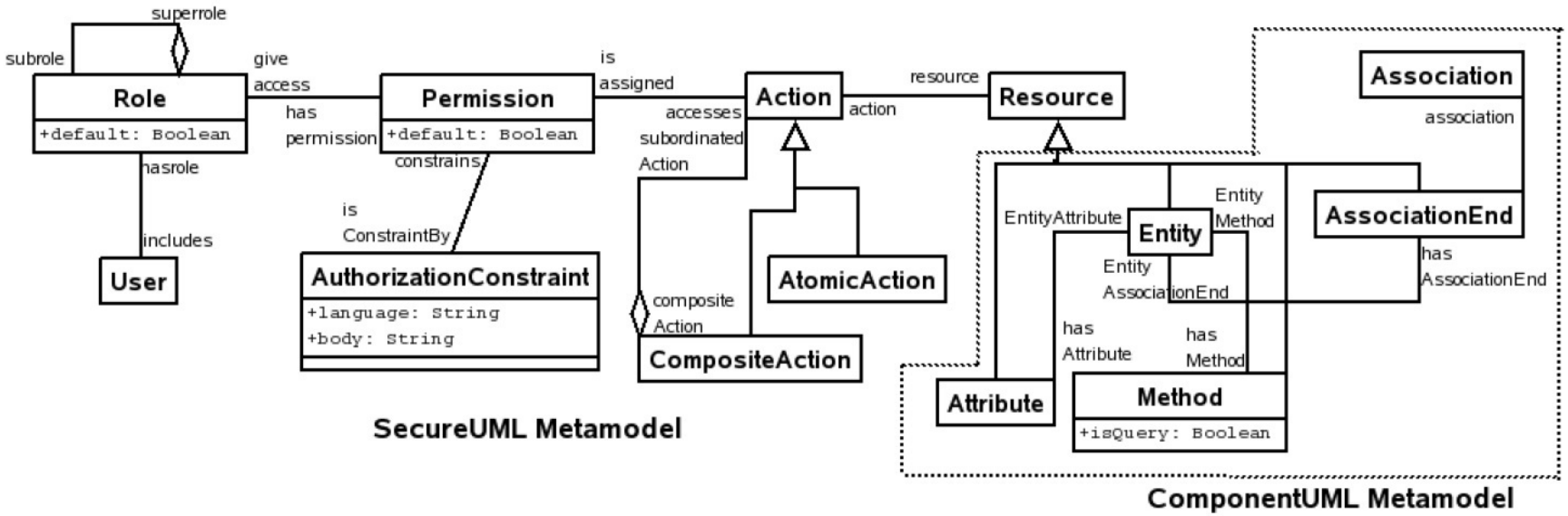
Metamodel. Conformance.

- A metamodel is a model composed of metadata whose elements formalize concepts and relations that define a modeling language.
- A model has conformance to a metamodel when it is written using the concepts and relations provided by the metamodel and it fulfill metamodel's constraints (if any).
  - E.g. Meeting is a model that has conformance to the ComponentUML metamodel.
  - E.g. SecureMeeting is a model that has conformance to the SecureUML+ComponentUML metamodel.

Example: Security Policy on the entity Meeting (CS)

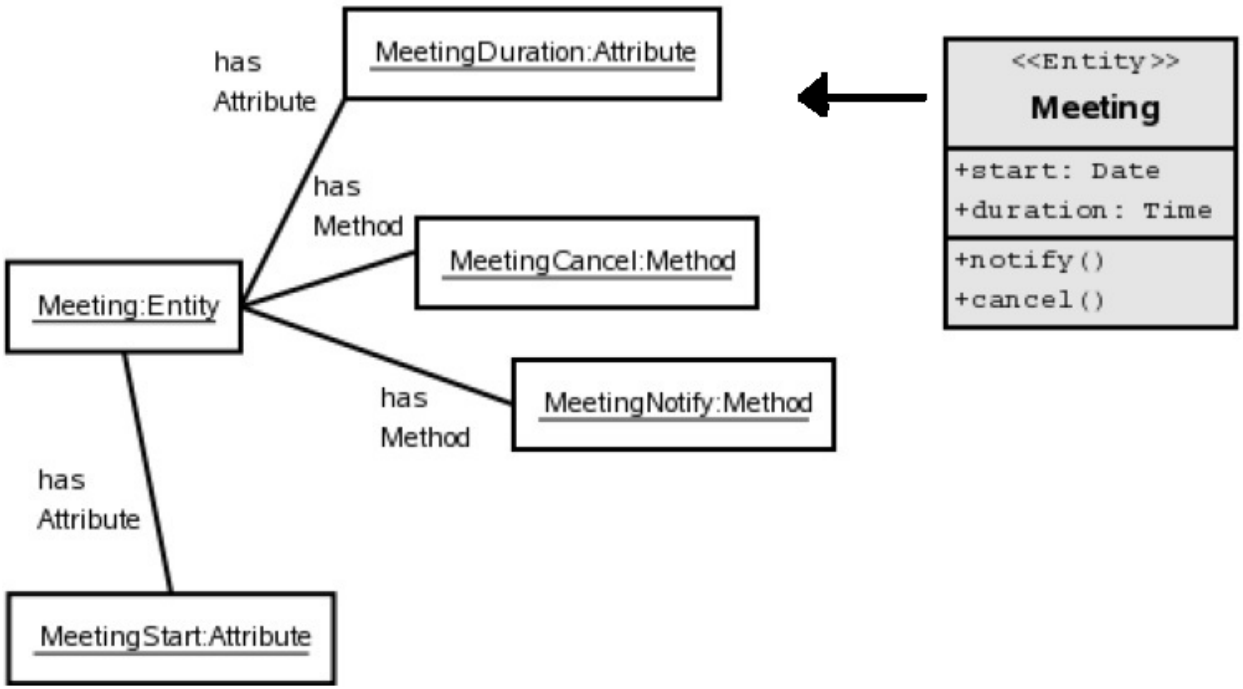


SecureUML+ComponentUML metamodel

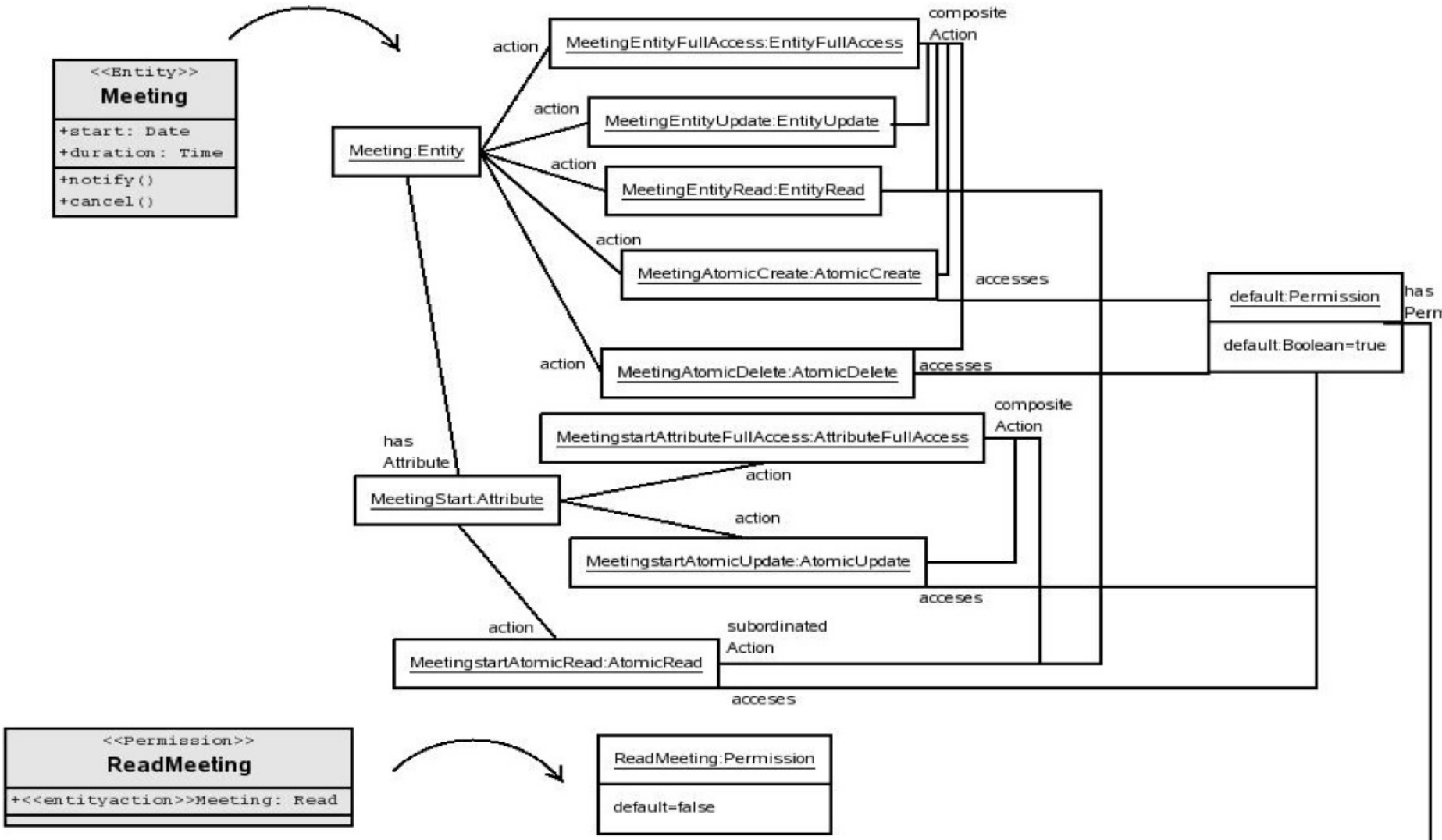


Meeting: Concrete to Abstract Syntax

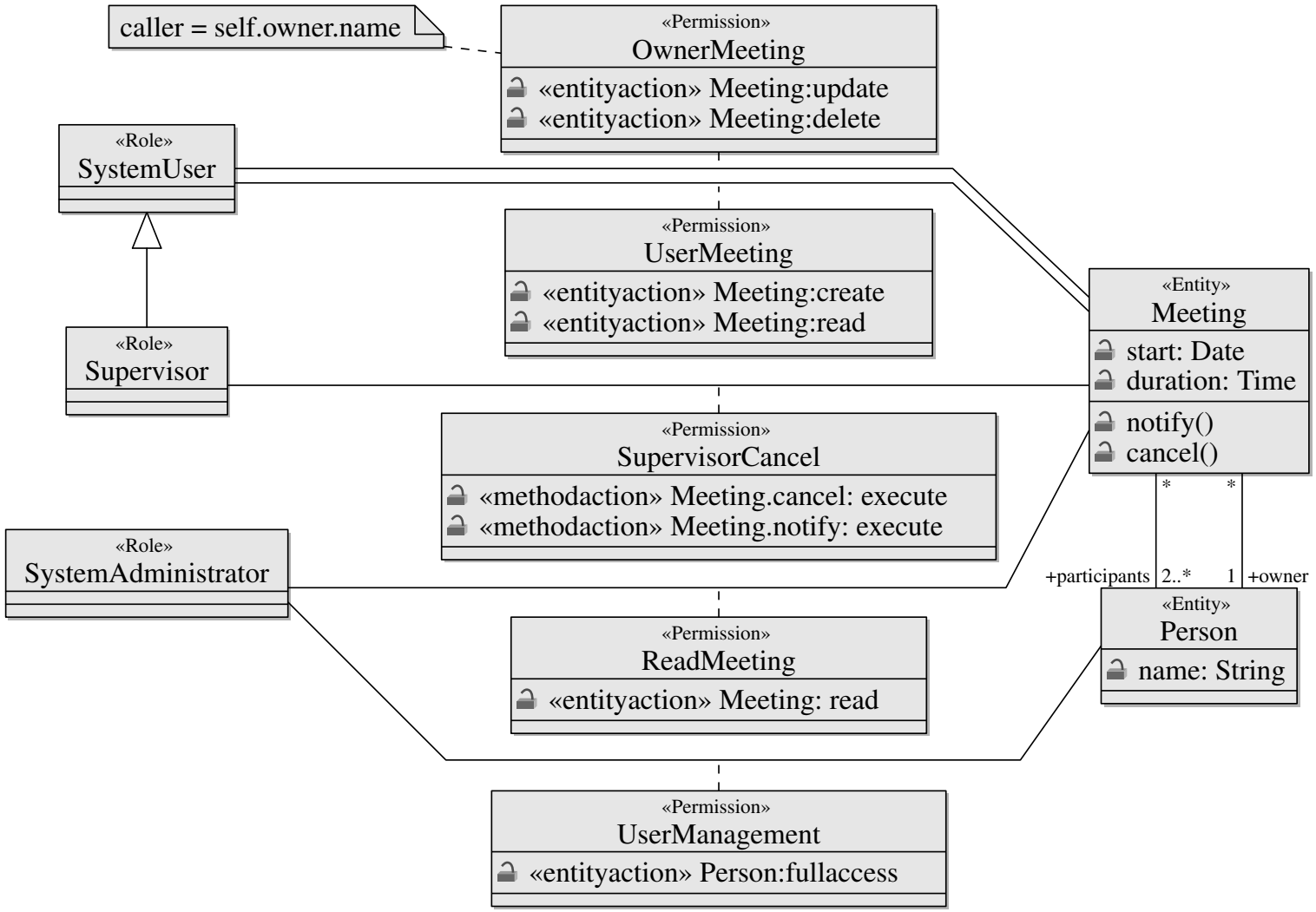
A model defined in concrete syntax can be map to abstract syntax as an instance of its metamodel.



Example: SecureMeeting Abstract Syntax



Example: Meeting-Scheduler security policy



Outline I

Type I: Questions about your security policy “per se”

- Are there a role ‘SystemAdministrator’?
- Has the role ‘Supervisor’ the permission ‘UserMeeting’ associated?
- Which are the actions associated to the permission ‘UserMeeting’ ?
- Is there a permission granting access to delete meetings?
- Can ‘Users’ edit meeting information?

Method for getting the answers to questions about your security-design models of Type I.

Type I. Questions about your security policy *"per se"*: e.g., are supervisors allowed to delete meetings?

- Step 1 Formalize your question as an OCL query, upon the SecureUML+ComponentUML metamodel (step manual, predefined library).
- Step 2 Evaluate your OCL query on the instance of the SecureUML+ComponentUML metamodel corresponding to your security-design model (automatic).

OCL queries to analyze security design models

- Which are the superroles of the ‘System Administrator’?, and of the ‘Supervisor’?
- It is not enough to run the query Supervisor.superrole to obtain the roles above a given role?

```
context Role::superrolePlus():Set(Role) body:
    self.superrolePlusOnSet(self.superrole)

context Role::superrolePlusOnSet(rs:Set(Role)):Set(Role) body:
 if rs.superrole)->exists(r|rs->excludes(r))
     then self.superrolePlusOnSet(rs ->union(rs.superrole)->asSet())
    else rs->including(self)
 endif
```

OCL queries to analyze security design models

- Which are the permissions of the role ‘Supervisor’?
- It is not sufficient to run the query Supervisor.hasPermission since permissions are inherited along the role hierarchy.

Supervisor.hasPermission —> {SupervisorCancel}

```
context Role::allPermissions():Set(Permission) body:
self.superrolePlus().haspermission->asSet()
```

Supervisor.allPermissions()
—> {SupervisorCancel, UserMeeting, OwnerMeeting}

OCL queries to analyze security design models

- subactionPlus() returns the collection of actions (directly or indirectly) subordinated to an action.

```
context Action::subactionPlus():Set(Action) body:
  if self.oclIsKindOf(AtomicAction)
     then Set{self}
     else self.oclAsType(CompositeAction)
        .subordinatedactions.subactionPlus()
  endif
```

- context Permission::allActions():Set(Action) body:
  self.accesses.subactionPlus()->asSet()
- context User::allAllowedActions():Set(Action) body:
  self.hasrole.allPermissions().allActions()->asSet()

OCL queries to analyze security design models

Supervisor.hasPermission.accesses —>
{Meeting:cancel:execute, Meeting:notify:execute}

Supervisor.allPermissions().accesses —>
{Meeting:create, Meeting:delete, Meeting:read,
Meeting:update, Meeting:cancel:execute,
Meeting:notify:execute }

Queries to analyze the security policy

- Do two permissions overlap? SystemUser, SystemAdministrator and Supervisor can read meeting information. They have overlapping permissions.

```
context Permission::overlapsWith(p:Permission):Boolean body:
self.allActions()->intersection(p.allActions())->notEmpty()
```

- Are there overlapping permissions for different roles? SystemAdministrator and Supervisor, SystemUser and SystemAdministrator.

```
context Permission::existOverlapping():Boolean body:
Permission.allInstances()->exists(p1,p2 | p1 <> p2 and
    p1.overlapsWith(p2) and
    not(p1.allRoles()->includesAll(p2.allRoles())))
```

Outline I

Semantic correspondence

Let $\Im_{RBAC}$ be the Σ_{RBAC} structure defined by a model M.
$\forall u \in Users$, $p \in Permissions$, and $a \in Actions$

is satisfied in $\Im_{RBAC}$	evaluates to true over $\overline{M}$
$UA(u, r)$	u.hasrole->includes(r)
$PA(r, p)$	r.haspermission->includes(p)
$AA(p, a)$	p.accesses->includes(a)

Semantic correspondence

is satisfied in $\Im_{RBAC}$	evaluates to true in $\overline{M}$
$r_1 \geq_{Roles} r_2$	r_2.superrolePlus()->includes(r_1) r_1.subrolePlus()->includes(r_2)
$\exists r_2 \in Roles.$ $r_2 \geq_{Roles} r_1 \wedge PA(r_2, p)$	r_1.allPermissions()->includes(p) p.allRoles()->includes(r_1)
$a_1 \geq_{Actions} a_2$	a_1.subactionPlus()->includes(a_2) a_2.compactionPlus()->includes(a_1)
$\exists a_2 \in Actions.$ $a_2 \geq_{Actions} a_1 \wedge AA(p, a_2)$	p.allActions->includes(a_1) a_1.allAssignedPermisssions() ->includes(p)
$\phi_{RBAC}(u, a)$	u.allAllowedActions()->includes(a)

Semantic correspondence

$\phi_{RBAC}(u, a)$: has the user u a permission to perform the action a?

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

Example: Queries on Meeting-Scheduler security policy

- Are there any users with the role ‘Administrator’?
  User.allInstances()->exists(u | u.hasRole.superrolePlus()->include(‘Administrator’))
- Has the role ‘Supervisor’ granted the permission ‘UserMeeting’?
  Supervisor.allPermissions()->includes(UserMeeting)
- Is the permission ‘UserMeeting’ granting access to execute the action ‘Meeting:Create’?
  UserMeeting.allActions()->includes(Meeting:Create)
- Is there a permission granting access to delete meetings?
  Permission.allInstances().accesses->includes(Meeting:Delete)

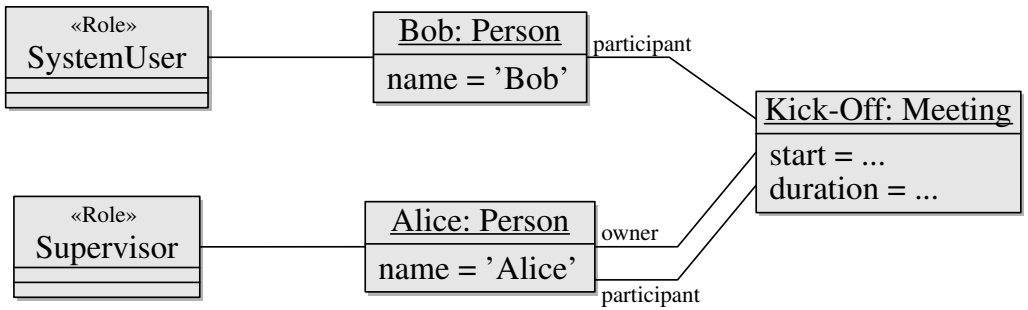
Outline I

Method for getting the answers to questions about your security-design models of Type II.

Type II. Questions about your security policy, but *applied to a concrete scenario*: e.g., can Bob delete the kick-off meeting of the NESSoS project?

- Step 1 For each of the roles assigned to the given user in the scenario, query your security policy about the conditions for users in that role to be allowed to perform the ("abstract" version of) given action. (manual, and follows the same technique explained before).
- Step 2 Evaluate the conditions obtained on the given scenario (automatic).

Example: Meeting-Scheduler security policy scenario



Queries to get the conditions

- Which is the collection of actions that are permitted for a given user?

```
context User::allAllowedActions():Set(Action) body:
    self.hasrole.allPermissions().allActions()->asSet()
```

- Given a user and an action, which authorization constraints must be satisfied for the user to perform this action?

```
context User::allAuthConstUser(a:Action):Set(AuthConstraint) body:
    self.hasrole.superrolePlus().allAuthConstRole(a)

context Role::allAuthConstRole(a:Action):Set(AuthConstraint) body:
self.permissionPlus(a).isconstraintby
```

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Queries to analyze the security policy scenario

- Can Bob cancel a meeting?
  Bob.allAllowedActions()->includes(Meeting:cancel:execute) –> false.
- Can Alice cancel a meeting?
  Alice.allAllowedActions()->includes(Meeting:cancel:execute) –> true.
- Can Bob update a meeting? Under which constraints?
  Bob.allAllowedActions()->includes(Meeting:update:execute) –> true.
  Bob.hasRole.allAuthConstRole(Meeting:update:execute) –> caller = self.owner
- Are there a users in the system allowed to create new meetings?
  Users.allInstances()->exists(u | u.allAllowedActions()->includes(Meeting:Create)) —> true

Outline I

Method for getting the answers to questions about your security-design models of Type III.

. . . *applied to scenarios in general*: e.g., can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own? Your question is whether there exists a scenario in which

- the authorization constraints for the given role to perform the given action (e.g., the "caller" being the owner of the meeting to be deleted) hold *AND*
- the invariants of your system hold *AND*
- your *additional contraints* (e.g.., the "caller" not being the owner of the meeting to be deleted) also hold.

Method for getting the answers to questions about your security-design models of Type III.

Type III. Questions about your security policy, but *applied to scenarios in general*: e.g., can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own?

- Step 1 Query your security policy about the conditions for users in the given role to be allowed to perform the given action. (manual, same technique explained before).
- Step 2 Formalize in OCL your additional contraints.
- Step 3 Translate into first-order logic the obtained conditions, your additional constraints, and the invariants of your system. This step is automatic.
- Step 4 Use standard first-order theorem-prover techniques to solve the problem; SMT solvers may give you the solution automatically.

Method for getting the answers to questions about your security-design models of Type III.

Can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own?

Step 1 Conditions:

Supervisor.allAuthConstRole(Meeting:delete) —>
caller = self.owner

Step 2 Additional constraints: that he/she does not own?

caller <> self.owner

Step 3 Translate conditions, additional constraints and invariants to FOL.

- Condition:
  $\forall(caller), \forall(self)(Supervisor(caller)\ and\ Meeting(self)\ and\ (caller = owner(self))$
- Constraint: $(caller \neq owner(self))$
- Invariants: $\emptyset$

Outline I

Benefits. Limitation.

- Check the unsatisfiability of (sets of) OCL expressions can be a powerful and practical tool in the hands of modelers: e.g., it will allows them to
  - Verify class invariants, by checking that they logically imply the expected constraints/properties;
  - Verify method preconditions, by checking that the class invariants do not logically imply their negations; and
  - Verify method postconditions, by checking that they do not logically imply the negation of (any of) the class invariants.
- Since OCL in full is undecidable, one can only expect to have an automated unsatisfiability checker for a large class of OCL expressions.

Unsatisfiability

Definition

Given a model (class diagram) $\mathcal{M}$, and a set of OCL constraints Φ, we say that Φ is $\mathcal{M}$-*unsatisfiable* if and only if there does not exist an $\mathcal{M}$-instance (object diagram) $\mathcal{O}$ on which every constraint in Φ evaluates to true

1

[1] We do not assume that instances always have a finite number of elements

Our mapping in a nutshell

Defined recursively over the structure of OCL expressions:

- Boolean-expressions are translated by formulas, which mirror their logical structure; Integer-expressions are basically copied;
- Collection-expressions are translated by predicates, whose meaning is defined by formulas also generated by the mapping;
- Association-ends are translated by predicates, also defined by formulas generated by the mapping;
- Attributes are translated by uninterpreted functions.

Association Ends

Given an association between two classes $Class_1$ and $Class_2$, with association-ends $AssocEnd^{Class_1}$ and $AssocEnd^{Class_2}$, the function $\mathrm{map}()$ generates the following sentences:

$$\forall(x,y)(AssocEnd^{Class_1}(x,y) \Rightarrow Class_1(y)).$$
$$\forall(x,y)(AssocEnd^{Class_2}(x,y) \Rightarrow Class_2(y)).$$
$$\forall(x,y)(AssocEnd^{Class_1}(x,y) \Leftrightarrow AssocEnd^{Class_2}(y,x)).$$

E.g.

$$\forall(x,y)(\mathrm{owner}(x,y) \Rightarrow \mathrm{Person}(y))$$
$$\forall(x,y)(\mathrm{ownedBy}(x,y) \Rightarrow \mathrm{Meeting}(y))$$
$$\forall(x,y)(\mathrm{owner}(x,y) \Leftrightarrow \mathrm{ownedBy}(y,x))$$

Some simple Mapping Rules

The function map() returns FOL sentences that mirror the logical structure of the original OCL.

$$\text{map}(\textit{BoolExpr} \text{ implies } \textit{BoolExpr}') = \{\text{make\_conj}(\text{map}(\textit{BoolExpr})) \Rightarrow \text{make\_conj}(\text{map}(\textit{BoolExpr}'))\}.$$

$$\text{map}(\textit{SetExpr}\text{->isEmpty}()) = \{\forall(x^{\flat})(\neg(\text{in\_coll}(\text{name}(\textit{SetExpr}), x^{\flat})))\} \cup \text{map}(\textit{SetExpr}).$$

$$\text{map}(\textit{SetExpr}\text{->includes}(\textit{ObjExpr})) = \{\text{in\_coll}(\text{name}(\textit{SetExpr}), \text{name}(\textit{ObjExpr}))\} \cup \text{map}(\textit{SetExpr}).$$

Examples

$$\text{map}(\text{Meeting.allInstances()->isEmpty()}) = \{\forall(x)(\neg\text{Meeting}(x))\}.$$

$$\text{map}(\text{Meeting.allInstances()->forAll(x|x.owner->isEmpty())}) = \{\forall(x)(\text{Meeting}(x) \Rightarrow \forall(y)(\neg(\text{owner}(x,y)\}.$$

$$\text{map}(\text{Meeting.allInstances()->exists(a|a.owner->notEmpty())}) = \{\exists(a)(\text{Meeting}(a) \wedge \exists(x)(\text{owner}(a,x)))\}.$$

More sophisticated Mapping Rules

$$\begin{aligned}
&\text{map}(\mathit{SetExpr}\text{->exists(x|}\mathit{BoolExpr}\text{))} \\
&\quad = \{\exists(x^{\flat})(\text{in_coll}(\text{name}(\mathit{SetExpr}), x^{\flat}) \\
&\qquad \wedge\ \text{make_conj}(\text{map}(\mathit{BoolExpr}[x \mapsto x^{\flat}]))))\} \\
&\quad \cup\ \text{map}(\mathit{SetExpr}).
\end{aligned}$$

$$\begin{aligned}
&\text{map}(\mathit{SetExpr}\text{->forAll(x|}\mathit{BoolExpr}\text{))} \\
&\quad = \{\forall(x^{\flat})(\text{in_coll}(\text{name}(\mathit{SetExpr}), x^{\flat}) \\
&\qquad \Rightarrow\ \text{make_conj}(\text{map}(\mathit{BoolExpr}[x \mapsto x^{\flat}]))))\} \\
&\quad \cup\ \text{map}(\mathit{SetExpr}).
\end{aligned}$$

$$\begin{aligned}
&\text{map}(\mathit{SetExpr}\text{->select(}x\text{|}\mathit{BoolExpr}\text{))} \\
&\quad = \{\forall(y^{\flat})(\text{in_coll}(\text{name}(\mathit{SetExpr}\text{->select(}x\text{|}\mathit{BoolExpr}[x \mapsto y^{\flat}])), y^{\flat}) \\
&\qquad \Leftrightarrow\ (\text{in_coll}(\mathit{name}(\mathit{SetExpr}), y^{\flat}) \wedge \\
&\qquad\quad \text{make_conj}(\text{map}(\mathit{BoolExpr}[x \mapsto y^{\flat}])))))\}.
\end{aligned}$$

Examples

$$\begin{aligned}
&\text{map}(\text{Meeting.allInstances()->exists(x |Meeting.allInstances()->excluding(x)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\text{->includes(x)))} \\
&\quad = \{\exists(x)(\text{Meeting}(x) \wedge \text{excluding1}(x) \\
&\qquad\qquad \wedge(\forall(y)((\text{Meeting}(y) \wedge y \neq x) \Leftrightarrow \text{excluding1}(y))))\}.
\end{aligned}$$

$$\begin{aligned}
&\text{map}(\text{Meeting.allInstances()->collect(x|x.owner)->asSet()} \\
&\quad \text{->exists(y|y.ownedBy->isEmpty()))} \\
&\quad = \{\exists(y)(\text{collect1}(y) \wedge \forall(x)(\neg(\text{owner}(y,x))), \\
&\qquad \forall(z)(\text{collect1}(z) \Leftrightarrow \exists(w)(\text{Meeting}(w) \wedge \text{ownedBy}(w,z))))\}.
\end{aligned}$$

Recall Method for Type III.

Type III. Questions about your security policy, but *applied to scenarios in general*: e.g., can a scenario exist in which a supervisor is allowed to delete a meeting that he/she does not own?

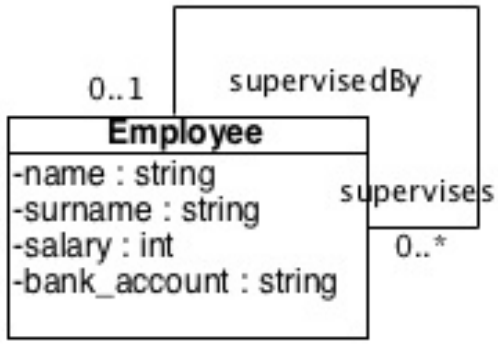
- Step 1 Query your security policy about the conditions for users in the given role to be allowed to perform the given action. (manual, same technique explained before).
- Step 2 Formalize in OCL your additional contraints.
- Step 3 Translate into first-order logic the obtained conditions, your additional constraints, and the invariants of your system. This step is automatic.
- Step 4 Use standard first-order theorem-prover techniques to solve the problem; SMT solvers may give you the solution automatically.

Outline I

ComponentUML example: Employee

Here we use ComponentUML to model the data associated with a company's employees.

- An employee has a name, a surname, a salary, and a bank account.
- An employee may possibly have a supervisor and may in turn supervise other employees.

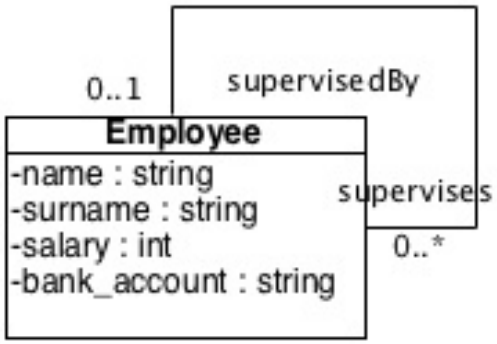


Employee model invariants

We can further specify this model by adding constraints to it.

For example, we can specify that:

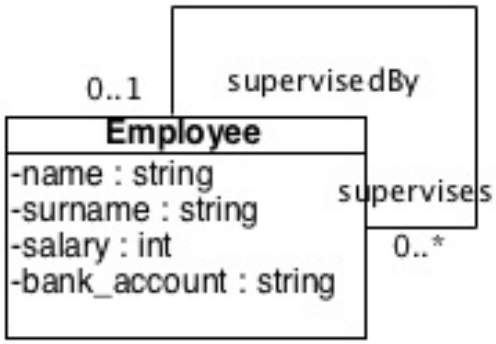
1. There is exactly one employee who has no supervisor.
   - Employee.allInstances()->one(ele.supervisedBy->isEmpty())
2. Nobody is his (or her) own supervisor.
   - Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))



Employee model invariants

We can further specify this model by adding constraints to it. For example, we can specify that:

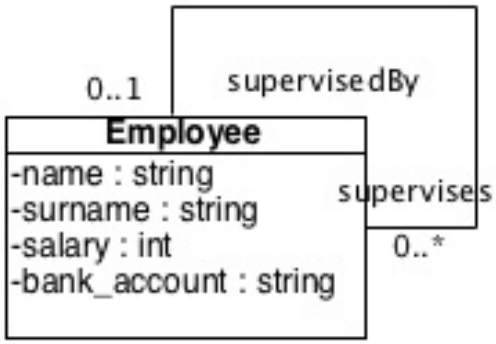
1. There is exactly one employee who has no supervisor.
   - Employee.allInstances()->one(ele.supervisedBy->isEmpty())
2. Nobody is his (or her) own supervisor.
   - Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))



Employee model invariants

We can further specify this model by adding constraints to it. For example, we can specify that:

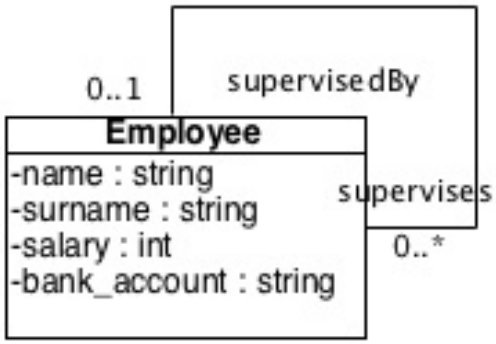
1. There is exactly one employee who has no supervisor.
   - Employee.allInstances()->one(ele.supervisedBy->isEmpty())
2. Nobody is his (or her) own supervisor.
   - Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))



Employee model invariants

We can further specify this model by adding constraints to it. For example, we can specify that:

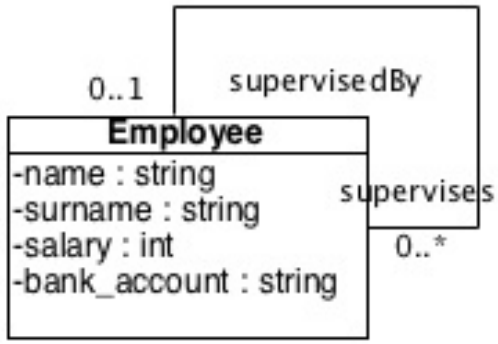
1. There is exactly one employee who has no supervisor.
   - Employee.allInstances()->one(ele.supervisedBy->isEmpty())
2. Nobody is his (or her) own supervisor.
   - Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))



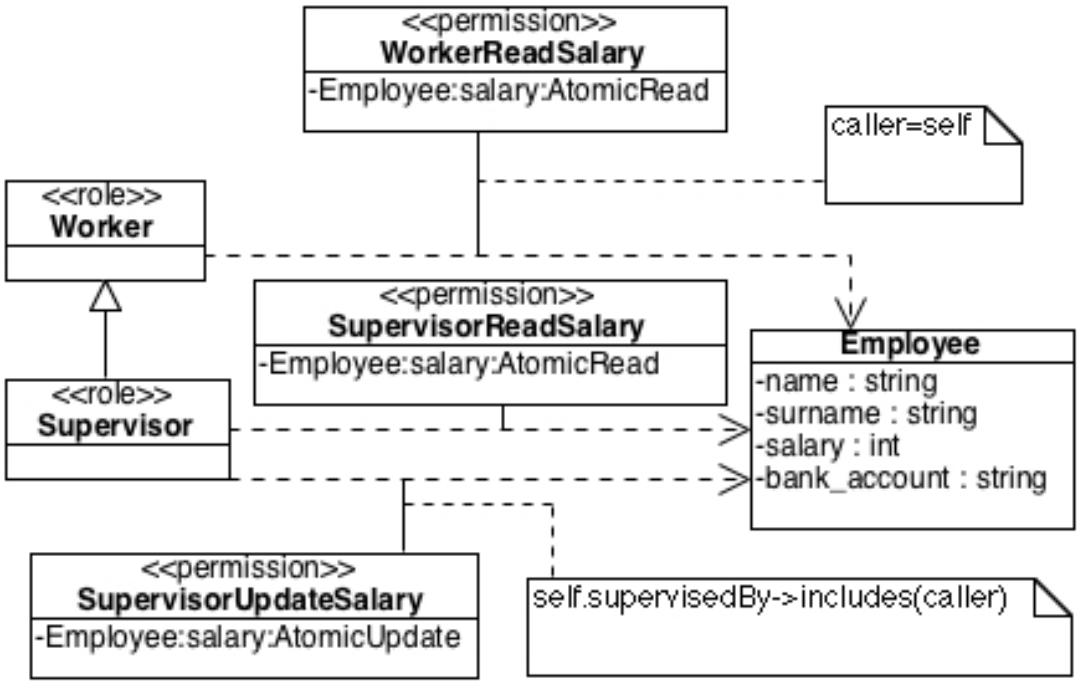
Employee model invariants

We can further specify this model by adding constraints to it. For example, we can specify that:

1. There is exactly one employee who has no supervisor.
   - Employee.allInstances()->one(ele.supervisedBy->isEmpty())
2. Nobody is his (or her) own supervisor.
   - Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))



Security-design model for Employees



SecureUML+ComponentUML example

Question:

Do our models allow an scenario in which a supervisor can change his/her own salary?

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
—-> unsat
```

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
—-> unsat
```

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
—–> unsat
```

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
—–> unsat
```

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
---> unsat
```

Model based analysis: OCLSat4Yices

```
(define Employee::(-> int bool))
(define supervisedBy::(-> int int bool))

; Cond:
Employee.allInstances()->exists(selflself.supervisedBy->includes(caller))
;Ad.Cons: caller=self

(assert (exists (self::int) (exists (caller::int)
    (and (Employee self) (and (Employee caller)
       (and (supervisedBy self caller) (= self caller)))))))

; Inv: Employee.allInstances()->forAll(ele.supervisedBy->excludes(e))

(assert (forall (x::int) (=> (Employee x) (not (supervisedBy x x)))))

(check)
—-> unsat
```

THANKS FOR YOUR ATTENTION. QUESTIONS?