Analyzing UML/OCL Models with HOL-OCL

Achim D. Brucker[1] Burkhart Wolff[2]

[1] SAP Research, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

[2] PCRI/co INRIA-Futurs, Parc Club Orsay Université, 91893 Orsay Cedex, France
wolff@lri.fr

A Tutorial at MoDELS 2008
Toulouse, 28th September 2008

Abstract

In this tutorial, we present the theorem proving environment HOL-OCL. The HOL-OCL system is an interactive proof environment for UML/OCL specifications that is integrated in a Model Driven Engineering (MDE) framework. HOL-OCL allows to reason over UML class models annotated with OCL specifications. Thus, HOL-OCL strengthens a crucial part of the UML to an object-oriented formal method. HOL-OCL provides several derived proof calculi that allow for formal derivations of validity of UML/OCL formulae. These formulae arise naturally when checking the consistency of class models, when formally refining abstract models to more concrete ones or when discharging side-conditions from model-transformations.

The latest version of these slides and all additional material is available at:
`http://projects/hol-ocl/2008-models-hol-ocl-tutorial/`

Outline

Outline

The Situation Today

A Software Engineering Problem

- Software systems
  - are becoming more and more complex and
  - are used in safety and security critical applications.
- Formal methods are one way to increase their reliability.
- But, formal methods are hardly used by mainstream industry:
  - difficult to understand notation
  - lack of tool support
  - high costs
- Semi-formal methods, especially UML,
  - are widely used in industry, but
  - they lack support for formal methodologies.

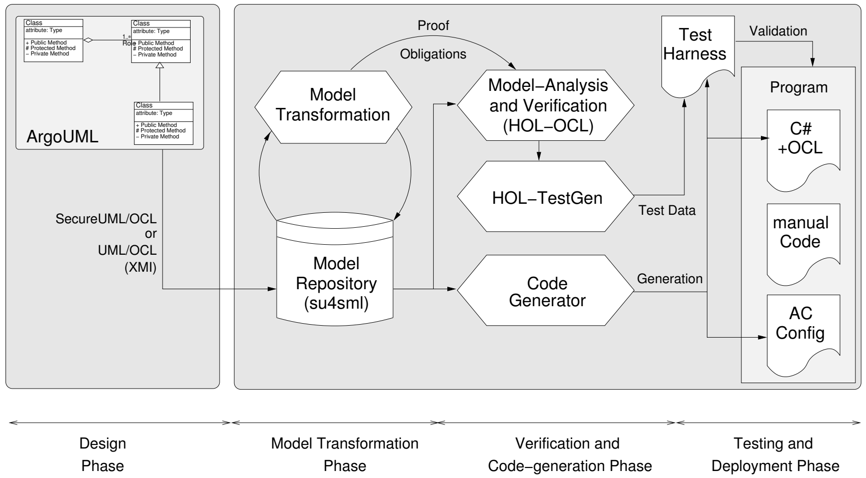
Is OCL an Answer?

- UML/OCL attracts the practitioners:
  - is defined by the object-oriented community,
  - has a “programming language face,”
  - increasing tool support.
- UML/OCL is attractive to researchers:
  - defines a “core language” for object-oriented modeling,
  - provides good target for object-oriented semantics research,
  - offers the chance for bringing formal methods closer to industry.

Turning OCL into a full-fledged formal methods is deserving and interesting.

The HOL-OCL Vision:

Tool Supported Formal Methods for (Model-driven) Software Development



ArgoUML

SecureUML/OCL or UML/OCL (XMI)

Model Transformation

Proof Obligations

Model–Analysis and Verification (HOL–OCL)

Model Repository (su4sml)

HOL–TestGen

Test Data

Test Harness

Validation

Program

C# +OCL

manual Code

AC Config

Code Generator

Generation

Design Phase

Model Transformation Phase

Verification and Code–generation Phase

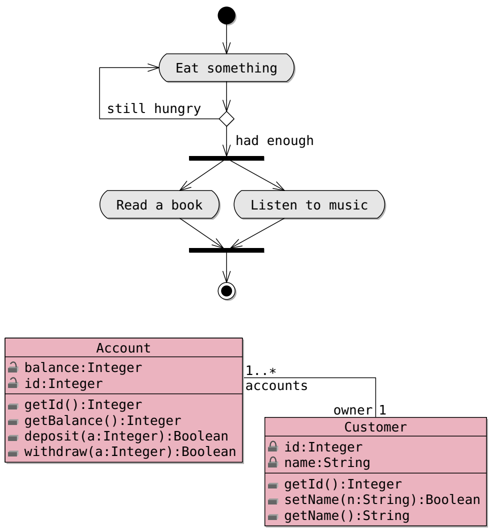
Testing and Deployment Phase

Outline

1. Introduction
2. **Background**
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

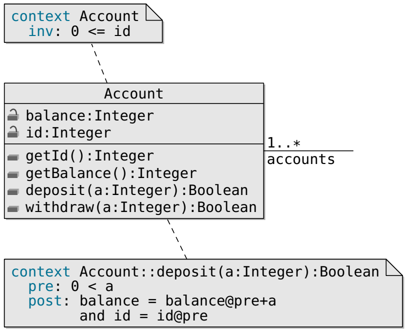
The Unified Modeling Language (UML)

- Visual modeling language
- Object-oriented development
- Industrial tool support
- OMG standard
- Many diagram types, e. g.,
  - activity diagrams
  - **class diagrams**
  - . . .



The Object Constraint Language (OCL)

- Textual extension of the UML
- Allows for annotating UML diagrams
- In the context of class–diagrams:
  - invariants
  - preconditions
  - postconditions
- Can be used for other diagrams



OCL by Example

- Class invariants:

```
context Account inv: 0 <= id
```

- Operation specifications:

```
context Account::deposit(a:Integer):Boolean
pre: 0 < a
post: balance = balance@pre + a
```

- A “uniqueness” constraint for the class `Account`:

```
context Account inv:
    Account::allInstances()
      ->forAll(a1,a2 | a1.id = a2.id implies a1 = a2)
```

OCL context OCL keywords UML path expressions

Formal Methods

> A **formal method** is a mathematically based technique
> for the specification, development and verification
> of software and hardware systems.

- A **strong formal method** is a formal method supported by formal tools, e. g., model-checkers or theorem provers.
- A **semi-formal method** lacks both, a sound formal definition of its semantics and support for formal tools.

Shallow vs. Deep Embeddings

Representing the logical operations *or* and *and* via a

- **shallow embedding:**

  Direct definition of the semantics, e.g. each construct is represented by some function on a semantic domain.

  $$x \text{ and } y \equiv \lambda e.\, x\, e \wedge y\, e \qquad x \text{ or } y \equiv \lambda e.\, x\, e \vee y\, e$$

- **deep embedding:**

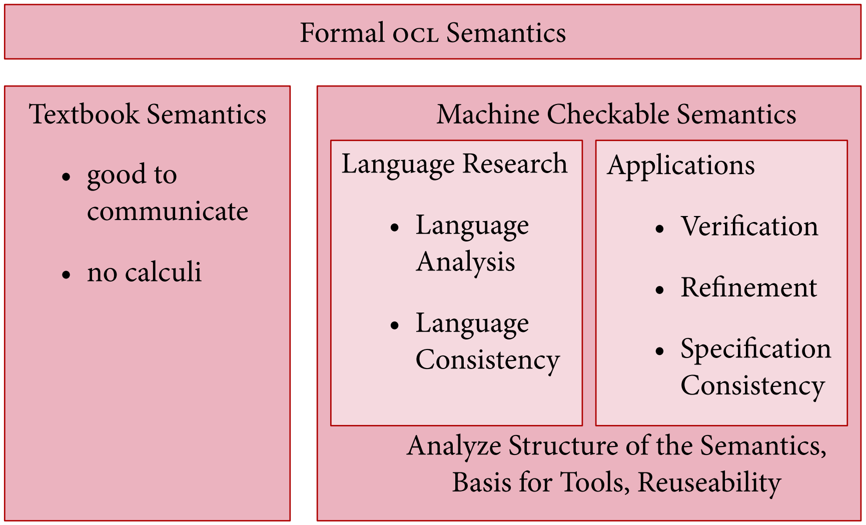
  The abstract syntax is presented as a datatype and a semantic function I from syntax to semantics.

  $$expr = \text{var } var \mid expr \text{ and } expr \mid expr \text{ or } expr$$

  and the explicit semantic function I:

  $$\begin{aligned} I[\![\text{var } x]\!] &= \lambda e\,.\, e(x) \\ I[\![x \text{and} y]\!] &= \lambda e\,.\, I[\![x]\!]\; e \wedge I[\![y]\!]\; e \\ I[\![x \text{or} y]\!] &= \lambda e\,.\, I[\![x]\!]\; e \vee I[\![y]\!]\; e \end{aligned}$$

Defining Semantics



Formal OCL Semantics

Textbook Semantics

- good to communicate
- no calculi

Machine Checkable Semantics

Language Research

- Language Analysis
- Language Consistency

Applications

- Verification
- Refinement
- Specification Consistency

Analyze Structure of the Semantics,
Basis for Tools, Reuseability

Textbook Semantics: Example

The Interpretation of the logical connectives:

b_1	b_2	b_1 and b_2	b_1 or b_2	b_1 xor b_2	b_1 implies b_2	not b_1
false	false	false	false	false	true	true
false	true	false	true	true	true	true
true	false	false	true	true	false	false
true	true	true	true	false	true	false
false	$\perp$	false	$\perp$	$\perp$	true	true
true	$\perp$	$\perp$	true	$\perp$	$\perp$	false
$\perp$	false	false	$\perp$	$\perp$	$\perp$	$\perp$
$\perp$	true	$\perp$	true	$\perp$	true	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Textbook Semantics: Summary

- Usually “Paper-and-Pencil” work in mathematical notation.
- Advantages
  - Useful to communicate semantics.
  - Easy to read.
- Disadvantages
  - No rules, no laws.
  - Informal or meta-logic definitions (*“The Set is the mathematical set.”*).
  - It is easy to write inconsistent semantic definitions.

Machine-checked Semantics: Example

Defining the core logic (Strong Kleene Logic):

$$
\begin{aligned}
\texttt{not}\ \_ &\equiv \mathrm{lift}_1\ \mathrm{strictify}(\lambda x.\ \llcorner\neg\ulcorner x\urcorner\lrcorner)\\
\_\ \texttt{and}\ \_ &\equiv \mathrm{lift}_2\ (\lambda x\, y.\ \text{if}\ (\mathrm{def}\, x)\\
&\qquad\qquad \text{then if}\ (\mathrm{def}\, y)\ \text{then}\ \llcorner\ulcorner x\urcorner \wedge \ulcorner y\urcorner\lrcorner\\
&\qquad\qquad\qquad\qquad \text{else if}\ \ulcorner x\urcorner\ \text{then}\ \bot\ \text{else}\ \llcorner \mathrm{false}\lrcorner\\
&\qquad\qquad \text{else if}\ (\mathrm{def}\, y)\ \text{then if}\ \ulcorner y\urcorner\ \text{then}\ \bot\\
&\qquad\qquad\qquad\qquad \text{else}\ \llcorner \mathrm{false}\lrcorner\ \text{else}\ \bot)\\
\_\ \texttt{or}\ \_ &\equiv \lambda x\, y.\ \texttt{not}\ (\texttt{not}\ x\ \texttt{and}\ \texttt{not}\ y)\\
\_\ \texttt{implies}\ \_ &\equiv \lambda x\, y.\ (\texttt{not}\ x)\ \texttt{or}\ \ y
\end{aligned}
$$

Machine-checked Semantics: Summary

Motivation: Honor the semantical structure of the language.

- A machine-checked semantics
  - conservative embeddings guarantee **consistency** of the semantics.
  - builds the basis for **analyzing** language features.
  - allows incremental changes of semantics.
- Many theorems, like “A`->union`$(B) = B$`->union`(A)” can be automatically lifted based on their HOL variants.
- As basis of further tool support for
  - **reasoning** over specifications.
  - **refinement** of specifications.
  - automatic **test data generation**.

Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

Developing Formals Tools for UML/OCL?

Turning UML/OCL into a formal method

1. A formal semantics of **UML class models**
   - typed path expressions
   - inheritance
   - dynamic binding
   - . . .
2. A formal semantics of **OCL** and proof support for OCL
   - reasoning over UML path expressions
   - large libraries
   - three-valued logic
   - . . .

Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
    - Formalization of OCL
    - Formalization of UML
    - The OCL Standard
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

How to Formalize OCL ?

The semantic foundation of the OCL standard:

Chapter 11 "The OCL Standard Library" (normative):
describes the requirements (pre-/post-style)

Appendix A "Semantics" (informative):
presents a formal semantics (paper and pencil)

The OCL Semantics: An Example

- The Interpretation of "`X->union(Y)`" for sets ("$X \cup Y$"):

$$I(\cup)(X,Y) \equiv \begin{cases} X \cup Y & \text{if } X \neq \perp \text{ and } Y \neq \perp, \\ \perp & \text{otherwise} \end{cases}$$

- This is a
    - **lifted** (sets can be undefined, denoted by $\perp$) and
    - **strict** (the union of undefined with anything is undefined)

  version of the union of "mathematical sets."

A Machine-checked Semantics

- Our formalization of "`X->union(Y)`" for sets ("$X \cup Y$"):

$$\_\texttt{->union}\_ \equiv \Big(\text{strictify}\big(\lambda X.\ \text{strictify}(\lambda Y.\ \llcorner \ulcorner X \urcorner \cup \ulcorner Y \urcorner \lrcorner)\big)\Big)\,.$$

- We model concepts like **strict** and **lifted** explicit, i. e., we introduce:
    - a datatype for lifting:

$$\alpha_\perp := \llcorner \alpha \lrcorner \mid \perp$$

    - a combinator for strictification:

$$\text{strictify}\ f\ x \equiv \text{if } x = \perp \text{ then } \perp \text{ else } f\ x$$

Is This Semantics Compliant?

- We prove formally (within our embedding):

$$\text{Sem}[\![\texttt{not}\ \ X]\!]\gamma = \begin{cases} \llcorner\neg\ulcorner\text{Sem}[\![X]\!]\gamma\urcorner\lrcorner & \text{if Sem}[\![X]\!]\gamma \neq \bot\,,\\ \bot & \text{otherwise}\,.\end{cases}$$

```
lemma "(Sem⟦not x⟧γ) = (if Sem⟦x⟧γ ≠ ⊥ then ⌞¬⌜Sem⟦x⟧γ⌝⌟ else ⊥)"
  apply(simp add: OclNot_def DEF_def lift0_def lift1_def lift2_def
                  semfun_def )
done
```

Proving Requirements

isEmpty() : Boolean (11.7.1-g)

Is self the empty collection?

`post: result = ( self->size() = 0 )`

Bag

```
lemma (self ->isEmpty()) = ((self, β :: bot)Bag)->size() ≐ 0
 apply(rule Bag_sem_cases_ext, simp_all)
  apply(simp_all add: OCL_Bag.OclSize_def OclMtBag_def
                      OclStrictEq_def
                      Zero_ocl_int_def ss_lifting')
  done
```

Outline

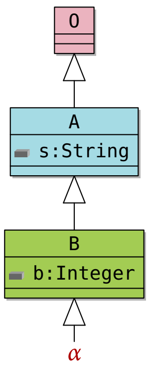
A Semantics of Typed Path Expressions

Question: What is the semantics of `self.s`?

Access the value of the attribute s of the object `self`.

- Formalizing **type safe** path expressions requires
  - a HOL representation of class types
  - HOL functions for accessing attributes
  - support for inheritance and subtyping
- After **adding new classes** to a model
  - there is no need for re-proving
  - definitions can be re-used
- Goal: a type-safe object store, supporting modular proofs

Representing Class Types



- The “extensible records” approach
  - We assume a common superclass (0).
  - The uniqueness is guaranteed by a *tag type*, e. g.:

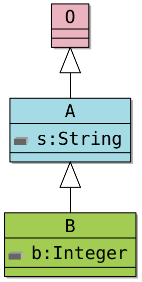
$$\mathrm{O}_{\mathrm{tag}} := \mathrm{classO}$$

  - Construct class type as tuple along inheritance hierarchy

$$\alpha\ \mathrm{B} := (\mathrm{O}_{\mathrm{tag}} \times \mathrm{oid}) \times \Big((\mathrm{A}_{\mathrm{tag}} \times \mathtt{String}) \times \big((\mathrm{B}_{\mathrm{tag}} \times \mathtt{Integer}) \times \alpha_\perp\big)_\perp\Big)_\perp$$

where $\_{}_\perp$ denotes types supporting undefined values.

Representing Class Types: Summary



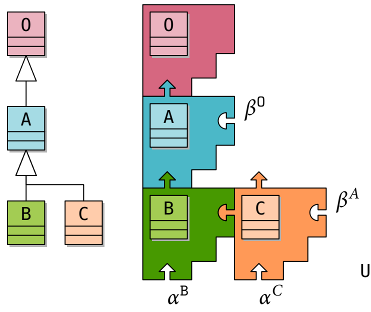
- Advantages:
  - it allows for extending class types (inheritance),
  - subclasses are type instances of superclasses
  - ⇒ **it allows for modular proofs**, i. e., a statement $\phi(x :: (\alpha\ \mathrm{B}))$ proven for class B is still valid after extending class B.
- However, it has a major disadvantage:
  - modular proofs are only supported for **one** extension per class

A Universe Type

A **universe** type represents all classes

- supports modular proofs with arbitrary extensions
- provides a formalization of a extensible typed object store

An Extensible Object Store



$$\mathrm{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

$$\mathrm{U}^1_{(\alpha^A,\beta^0)} = O \times (A \times \alpha^A_\perp + \beta^0)_\perp$$

$$\mathrm{U}^2_{(\alpha^B,\beta^0,\beta^A)} = O \times (A \times (B \times \alpha^B_\perp + \beta^A)_\perp + \beta^0)_\perp$$

$$\mathrm{U}^3_{(\alpha^B,\alpha^C,\beta^0,\beta^A)} = O \times (A \times (B \times \alpha^B_\perp + (C \times \alpha^C_\perp + \beta^A))_\perp + \beta^0)_\perp$$

$$\mathscr{U}^3_{(\alpha^B,\alpha^C,\beta^0,\beta^A)} \prec \mathscr{U}^2_{(\alpha^B,\beta^0,\beta^A)} \prec \mathscr{U}^1_{(\alpha^A,\beta^0)} \prec \mathscr{U}^0_{(\alpha^0)}$$

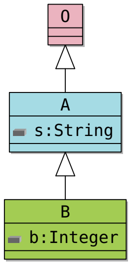
Operations Accessing the Object Store

- injections
  $\mathrm{mk}_{\mathrm{O}}\, o = \mathrm{Inl}\, o$ with type $\alpha^{O}\ 0 \to \mathscr{U}^{0}_{\alpha^{O}}$
- projections
  $\mathrm{get}_{\mathrm{O}}\, u = u$ with type $\mathscr{U}^{0}_{\alpha^{O}} \to \alpha^{O}\ 0$
- type casts
  $\mathrm{A}_{[\mathrm{O}]} = \mathrm{get}_{\mathrm{O}} \circ \mathrm{mk}_{\mathrm{A}}$ with type $\alpha^{A}\ \mathrm{A} \to (A \times \alpha^{A}_{\perp} + \beta^{O})\ 0$
  $\mathrm{O}_{[\mathrm{A}]} = \mathrm{get}_{\mathrm{A}} \circ \mathrm{mk}_{\mathrm{O}}$ with type $(A \times \alpha^{A}_{\perp} + \beta^{O})\ 0 \to \alpha^{A}\ \mathrm{A}$
- . . .

All definitions are generated automatically

Does This Really Model Object-orientation?

For each UML model, we have to show several properties:



- subclasses are of the superclasses kind:
  $$\frac{\mathrm{isType}_{\mathrm{B}}\ \mathit{self}}{\mathrm{isKind}_{\mathrm{A}}\ \mathit{self}}$$
- “re-casting”:
  $$\frac{\mathrm{isType}_{\mathrm{B}}\ \mathit{self}}{\mathit{self}_{[\mathrm{A}][\mathrm{B}]} \neq \perp \wedge \mathrm{isType}_{\mathrm{B}}\ (\mathit{self}_{[\mathrm{A}][\mathrm{B}][\mathrm{A}]})}$$
- monotonicity of invariants, . . .

All rules are derived automatically

First Results of Formalizing the OCL Standard

- We found several glitches:
  - inconsistencies between the formal semantics and the requirements
  - missing pre- and postconditions
  - wrong (e.g., to weak) pre- and postconditions
  - . . .
- and examined possible extensions (open problems):
  - operations calls and invocations
  - smashing of datatypes
  - equalities
  - recursion
  - semantics for invariants (type sets)
  - . . .

Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

Motivation

Observation:

- UML/OCL is a *generic* modeling language:
    - usually, only a sub-set of UML is used and
    - per se there is no standard UML-based development process.
- Successful use of UML usually comprises
    - a well-defined development process and
    - tools that integrate into the development process.

Conclusion:

- Formal methods for UML-based development should
    - support the local UML development methodologies and
    - integrate smoothly into the local toolchain.

A toolchain for formal methods should provide
tool-support for **methodologies**.

Well-formedness of Models

Well-formedness Checking

- Enforce **syntactical** restriction on (valid) UML/OCL models.
- Ensure a minimal quality of models.
- Can be easily supported by fully-automatic tools.

Example

- There should be at maximum five inheritance levels.
- The Specification of public operations may only refer to public class members.
- . . .

Proof Obligations for Models

Proof Obligation Generation

- Enforce **semantical** restriction on (valid) UML/OCL models.
- Build the basis for formal development methodologies.
- Require formal tools (theorem prover, model checker, etc).

Example

- Liskov's substitution principle.
- Model consistency
- Refinement.
- . . .

Proof Obligations: Liskov's Substitution Principle

Liskov substitution principle

Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T.

For constraint languages, like OCL, this boils down to:

- *pre-conditions* of overridden methods must be *weaker*.
- *post-conditions* of overridden methods must be *stronger*.

Which can formally expressed as implication:

- Weakening the pre-condition:

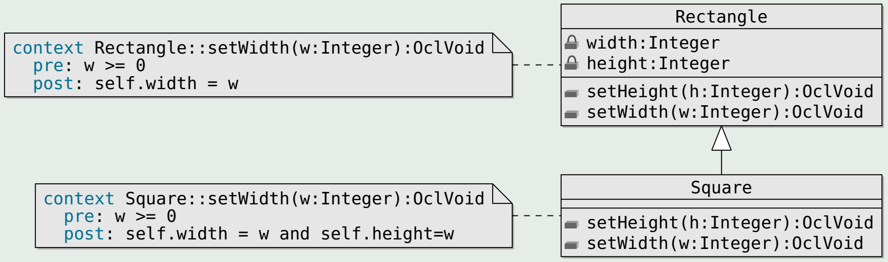
$$op_{\text{pre}} \longrightarrow op_{\text{pre}}^{\text{sub}}$$

- Weakening the pre-condition:

$$op_{\text{post}}^{\text{sub}} \longrightarrow op_{\text{post}}$$

Proof Obligations: Liskov's Substitution Principle

Example



```
context Rectangle::setWidth(w:Integer):OclVoid
  pre: w >= 0
  post: self.width = w
```

```
context Square::setWidth(w:Integer):OclVoid
  pre: w >= 0
  post: self.width = w and self.height=w
```

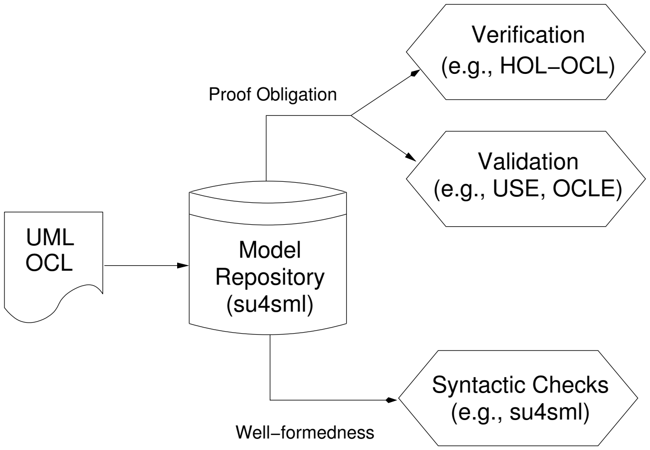
- Weakening the pre-condition:

$$(\texttt{w >= 0}) \longrightarrow (\texttt{w >= 0})$$

- Strengthening the post-condition:

$$(\texttt{self.width = w and self.height = w}) \longrightarrow (\texttt{self.width = w})$$

Well-formedness and Proof Obligations



Methodology

A tool-supported methodology should

- integrate into existing toolchains and processes,
- provide a unified approach, integrating ,
  - syntactic requirements (well-formedness checks),
  - generation of proof obligations,
  - means for **verification** (proving) or **validation**, and of course
- all phases should be supported by tools.

Example

A package-based object-oriented refinement methodology.

Refinement – Motivation

Support top-down development from an abstract model to a more concrete one.

- We start with an abstract transition system

$$sys_{\text{abs}} = (\sigma_{\text{abs}}, init_{\text{abs}}, op_{\text{abs}})$$

- We refine each abstract operation op_{abs} to a more concrete one: op_{conc}.
- Resulting in a more concrete transition system

$$sys_{\text{conc}} = (\sigma_{\text{conc}}, init_{\text{conc}}, op_{\text{conc}})$$

- Such refinements can be chained:

$$sys_1 \leadsto sys_2 \leadsto \cdots \leadsto sys_n$$

E.g., from an abstract model to one that supports code generation.

Refinement: Well-formedness

If package *B* refines a package *A*, then
one should be able to substitute every usage of package *A* with package *B*.

- The concrete package must provide at a corresponding public class for each public class of the abstract model.
- For public attributes we require that their type and for public operations we require that the return type and their argument types are either basic datatypes or public classes.
- For each public class of the abstract package, we require that the corresponding concrete class provides at least
  - public attributes with the same name and
  - public operations with the same name.
- The types of corresponding abstract and concrete attributes and operations are compatible.

Refinement: Proof Obligtations – Consistency

A transition system is consistent if:

- The set of initial states is non-empty, i. e.,

$$\exists \sigma.\ \sigma \in \mathit{init}$$

- The state invariant is satisfiable, i. e.,
  the conjunction of all invariants is invariant-consistent:

$$\exists \sigma.\ \sigma \models \mathit{inv}_1 \land \exists \sigma.\ \sigma \models \mathit{inv}_2 \land \cdots \land \exists \sigma.\ \sigma \models \mathit{inv}_n$$

- All operations op are implementable, i. e.,
  for each satisfying pre-state there exists a satisfying post-state:

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

Refinement: Proof Obligtations – Implements

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$ relating a concrete state S and an abstract states T.
- A forward refinement $S \sqsubseteq^{R}_{FS} T \equiv po_1(S,R,T) \land po_2(S,R,T)$ requires two proof obligations po_1 and po_2.
- **Preserve Implementability (po_1):**

$$\begin{tikzcd} \sigma_a \arrow[r, dashed, "op_a"] \arrow[d, leftrightarrow, "R"] & \sigma'_a \\ \sigma_c & \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \sigma_a \arrow[r, dashed, "op_a"] \arrow[d, leftrightarrow, "R"] & \sigma'_a \\ \sigma_c \arrow[r, dashed, "op_c"] & \sigma'_c \end{tikzcd}$$

$$po_1(S,R,T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.\ (\sigma_a, \sigma_c) \in R \rightarrow \sigma_c \in \text{pre}(T)$$

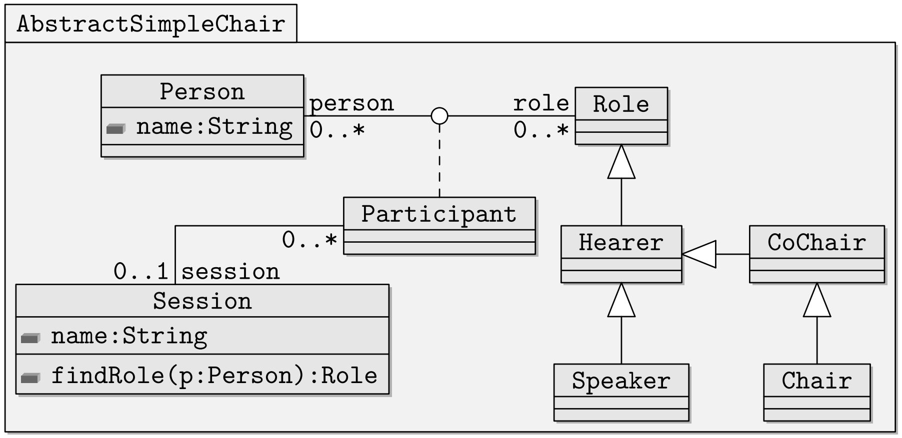
Refinement: Proof Obligtations – Refines

- Given an abstraction relation $R : \mathbb{P}(\sigma_{\text{abs}} \times \sigma_{\text{conc}})$ relating a concrete state S and an abstract states T.
- A forward refinement $S \sqsubseteq^{R}_{FS} T \equiv po_1(S,R,T) \land po_2(S,R,T)$ requires two proof obligations po_1 and po_2.
- **Refinement (po_2):**

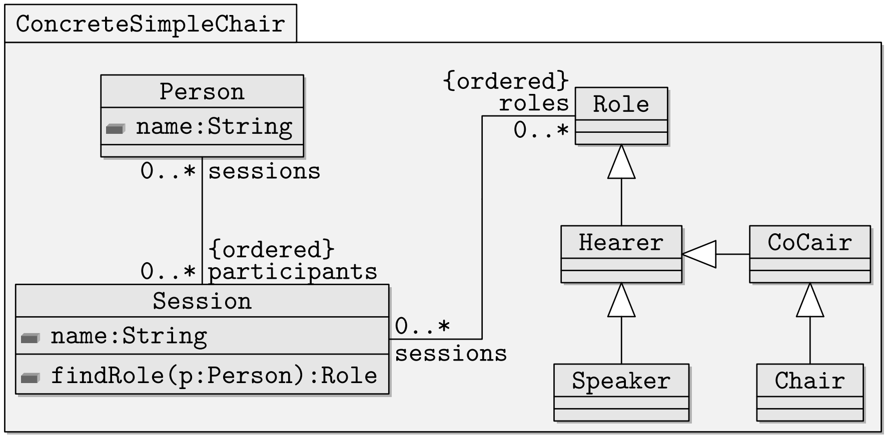
$$\begin{tikzcd} \sigma_a \arrow[d, leftrightarrow, "R"] & \\ \sigma_c \arrow[r, dashed, "op_c"] & \sigma'_c \end{tikzcd} \quad \Rightarrow \quad \begin{tikzcd} \sigma_a \arrow[r, dashed, "op_a"] \arrow[d, leftrightarrow, "R"] & \sigma'_a \arrow[d, leftrightarrow, dashed, "R"] \\ \sigma_c \arrow[r, dashed, "op_c"] & \sigma'_c \end{tikzcd}$$

$$po_2(S,R,T) \equiv \forall \sigma_a \in \text{pre}(S), \sigma_c \in V.\ \sigma_{c'}.\ (\sigma_a, \sigma_c) \in R \land (\sigma_c, \sigma'_c) \models_M T \rightarrow \exists \sigma'_a \in V.\ (\sigma_a, \sigma'_a) \models_M S \land (\sigma_{a'}, \sigma_{c'}) \in R$$

Refinement Example: Abstract Model



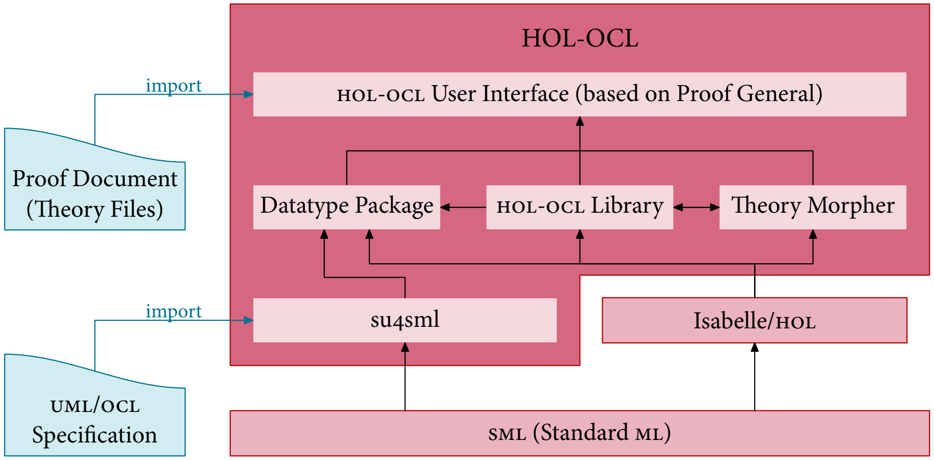
Refinement Example: Concrete Model



Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. **The HOL-OCL Architecture**
6. Applications
7. Conclusion and Future Work

The HOL-OCL Architecture



su4sml – Overview

su4sml is a UML/OCL (and SecureUML) model repository providing

- a database for syntactic elements of UML core, namely class models and state machines as well as OCL expressions.
- support for SecureUML.
- import of UML/OCL models in different formats:
    - XMI and ArgoUML (class models and state machines)
    - OCL (plain text files)
    - USE (plain text files describing class models with OCL annotations)
- a template-based code generator (export) mechanism.
- an integrated framework for model transformations.
- a framework for checking well-formedness conditions.
- a framework for generating proof obligations.
- an interface to HOL-OCL (encoder, po manager).

---

su4sml – Code Generators

su4sml provides a template-based code generator for

- Java, supporting
    - class models and state machines
    - OCL runtime enforcement
    - SecureUML
- C#, supporting
    - class models and state machines
    - SecureUML
- USE
- . . .

---

su4sml – Model Transformations

su4sml provides a framework for model transformation that

- supports the generation of proof obligations
- can be programmed in SML.

Currently, the following transformations are provided:

- a family of semantic preserving transformations for converting associations (e. g., *n*-ary into binary ones)
- a transformation from SecureUML/ComponentUML to UML/OCL.

---

su4sml – Well-formedness Checks

su4sml provides an framework for extended well-formedness checking:

- Checks if a given model satisfies certain syntactic constraints,
- Allows for defining dependencies between different checks
- Examples for well-formedness checks are:
    - restricting the inheritance depth
    - restringing the use of private class members
    - checking class visibilities with respect to member visibilities
    - . . .
- Can be easily extended (at runtime).
- Is integrated with the generation of proof obligations.

su4sml – Proof Obligation Generator

su4sml provides an framework for proof obligation generation:

- Generates proof obligation in OCL plus minimal meta-language.
- Only minimal meta-language necessary:
  - Validity: ⊨ _, _ ⊨ _
  - Meta level quantifiers: ∃_. _, ∃_. _
  - Meta level logical connectives: _ ∨ _, _ ∧ _, ¬_
- Examples for proof obligations are:
  - (semantical) model consistency
  - Liskov’s substitution principle
  - refinement conditions
  - . . .
- Can be easily extended (at runtime).
- Builds, together with well-formedness checking, the basis for tool-supported methodologies.

The Encoder

The model encoder is the main interface between su4sml and the Isabelle based part of HOL-OCL. The encoder

- declarers HOL types for the classifiers of the model,
- encodes
  - type-casts,
  - attribute accessors, and
  - dynamic type and kind tests implicitly declared in the imported data model,
- encodes the OCL specification, i. e.,
  - class invariants
  - operation specifications

  and combines it with the core data model, and
- proves (automatically) methodology and analysis independent properties of the model.

The Library

The HOL-OCL library

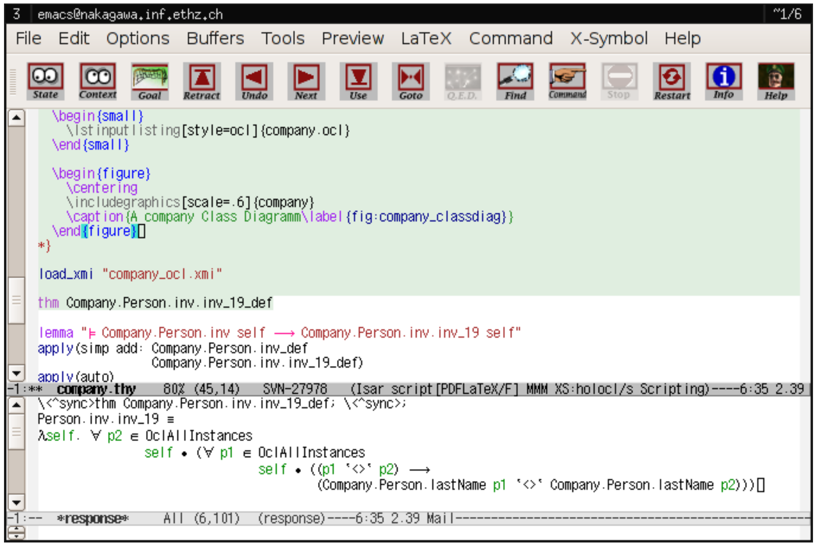
- formalizes the built-in operations of UML/OCL,
- comprises over 10 000 definitions and theorems,
- build the basis for new, OCL specific, proof procedures,
- provides proof support for (formal) development methodologies.

Tactics (Proof Procedures)

- OCL, as logic, is quite different from HOL (e. g., three-valuedness)
- Major Isabelle proof procedures, like `simp` and `auto`, cannot handle OCL efficiently.
- HOL-OCL provides several UML/OCL specific proof procedures:
  - embedding specific tactics (e. g., unfolding a certain level)
  - a OCL specific context-rewriter
  - a OCL specific tableaux-prover
  - . . .

  These language specific variants increase the degree of proof for OCL.

The HOL-OCL User Interface



```
  \begin{small}
    \lstinputlisting[style=ocl]{company.ocl}
  \end{small}

  \begin{figure}
    \centering
    \includegraphics[scale=.6]{company}
    \caption{A company Class Diagramm\label{fig:company_classdiag}}
  \end{figure}
*}

load_xmi "company_ocl.xmi"

thm Company.Person.inv.inv_19_def

lemma "⊨ Company.Person.inv self ⟶ Company.Person.inv.inv_19 self"
apply(simp add: Company.Person.inv_def
                Company.Person.inv.inv_19_def)
apply(auto)
-1:** company.thy    80% (45,14)    SVN-27978    (Isar script[PDFLaTeX/F] MMM XS:holocl/s Scripting)----6:35 2.39
```

```
\<^sync>thm Company.Person.inv.inv_19_def; \<^sync>;
Person.inv.inv_19 ≡
λself. ∀ p2 ∈ OclAllInstances
              self • (∀ p1 ∈ OclAllInstances
                          self • ((p1 `<>` p2) ⟶
                                (Company.Person.lastName p1 `<>` Company.Person.lastName p2)))
-1:-- *response*    All (6,101)    (response)----6:35 2.39 Mail-----------------------------------------------
```

The HOL-OCL High-level Language

The HOL-OCL proof language is an extension of Isabelle's Isar language:

- importing UML/OCL:

  ```
  import_model "SimpleChair.zargo" "AbstractSimpleChair.ocl"
              include_only "AbstractSimpleChair"
  ```

- check well-formedness and generate proof obligations for refinement:

  ```
  analyze_consistency [data_refinement] "AbstractSimpleChair"
  ```

- starting a proof for a generated proof obligation:

  ```
  po "AbstractSimpleChair.findRole_enabled"
  ```

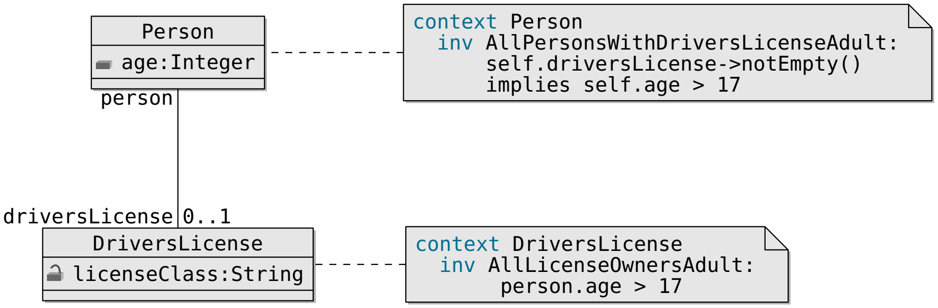
- generating code:

  ```
  generate_code "java"
  ```

Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

Simple Consistency Analysis I



```
context Person
  inv AllPersonsWithDriversLicenseAdult:
      self.driversLicense->notEmpty()
      implies self.age > 17
```

```
context DriversLicense
  inv AllLicenseOwnersAdult:
      person.age > 17
```

Figure: A simple model of vehicles and licenses

Simple Consistency Analysis II

```
lemma
assumes "τ ⊨ (Vehicles.Person.driversLicense(
                Vehicles.DriversLicense.person self)).IsDefined()"
                and "τ ⊨ (Vehicles.Person.age
                       (Vehicles.DriversLicense.person self)).IsDefined() "
  shows "τ ⊨ Person.inv.AllPersonsWithDriversLicenseAdult (
                  Vehicles.DriversLicense.person self)
             ⟶ τ ⊨ DriversLicense.inv.AllLicenseOwnersAdult self"
  apply(auto elim!: OclImpliesE)
  apply(cut_tac prems)
  apply(auto simp: inv.AllPersonsWithDriversLicenseAdult_def
                   inv.AllLicenseOwnersAdult_def
            elim!: OclImpliesE SingletonSetDefined)
done
```

Liskov's Substitution Principle I

```
context A::m(p:Integer):Integer
  pre: p > 0
  post: result > 0

context A::m(p:Integer):Integer
  pre: p >= 0
  post: result = p*p + 5

-- The following constraints overrides the specification for
-- m(p:Integer):Integer that was originally defined in
-- class A, i.e., C is a subclass of A.
-- (Stricly, this is not valid with respect to the
--  UML/OCL standards...)
context C::m(p:Integer):Integer
  pre:  p >= 0
  post: result > 1 and result = p*p+5
```

Liskov's Substitution Principle II

```
import_model "overriding.zargo" "overriding.ocl"

generate_po_liskov "pre"
generate_po_liskov "post"

po "overriding.OCL_liskov−po_lsk_pre−1"
  apply(simp add: A.m_Integer_Integer.pre1_def
                  A.m_Integer_Integer.pre1.pre_0_def
                  C.m_Integer_Integer.pre1_def
                  C.m_Integer_Integer.pre1.pre_0_def
                  A.m_Integer_Integer.pre1.pre_1_def)
  apply(ocl_auto)
discharged
```

Outline

1. Introduction
2. Background
3. Formalization of UML and OCL
4. Mechanized Support for Model Analysis Methods
5. The HOL-OCL Architecture
6. Applications
7. Conclusion and Future Work

Conclusion

- HOL-OCL provides:
  - a formal, machine-checked semantics for OO specifications,
  - an interactive proof environment for OO specifications,
  - publicly available:
    `http://www.brucker.ch/projects/hol-ocl/`,
  - next (major) release planned in October/November 2008.
- HOL-OCL is integrated into a toolchain providing:
  - extended well-formedness checking,
  - proof-obligation generation,
  - methodology support for UML/OCL,
  - a transformation framework (including PO generation),
  - code generators,
  - support for SecureUML.

Ongoing and Future Work

- Ongoing work includes improving the infrastructures for
  - well-formedness-checking,
  - proof-obligation generation (Liskov, Refinement,),
  - consistency checking,
  - Hoare-style program verification,
  - better proof automation in general.
- Future works could include the development for
  - integrating OCL validation tools, e.g., USE,
  - test-case generation (i.e., integrating HOL-TestGen),
  - supporting SecureUML.
  - . . . .

Thank you
for your attention!

Any questions or remarks?

Bibliography I

Achim D. Brucker, Jürgen Doser, and Burkhart Wolff.
An MDA framework supporting OCL.
Electronic Communications of the EASST, 5, 2006.

Achim D. Brucker.
An Interactive Proof Environment for Object-oriented Specifications.
Ph.d. thesis, ETH Zurich, March 2007.
ETH Dissertation No. 17097.

Achim D. Brucker and Burkhart Wolff.
HOL-OCL – A Formal Proof Environment for UML/OCL.
In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering* (FASE08), number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, 2008.

Bibliography II

Achim D. Brucker and Burkhart Wolff.
Extensible universes for object-oriented data models.
In Jan Vitek, editor, ECOOP *2008 – Object-Oriented Programming*, number 5142 in Lecture Notes in Computer Science, pages 438–462. Springer-Verlag, 2008.

The HOL-OCL Website.
`http://www.brucker.ch/projects/hol-ocl/`.

Part II

Appendix

Outline

8 SecureUML – Model-driven Security

Outline

8 SecureUML – Model-driven Security
  - SecureUML
  - A Formal Model Transformation
  - Consistency Analysis

Model-driven Security

Goals:

- A method to model secure designs and automatically transform these into secure systems.
- Supports well-established standards/technology for modelling components and security.
- Models are expressive, comprehensible, and maintainable.
- Reduces complexity of application development and improves the quality of the resulting applications.
- The entire process is semantically well-founded.
- Allows integrated formal reasoning over security design models.

SecureUML

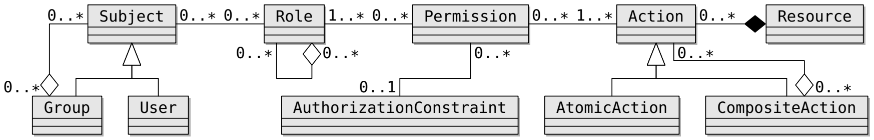


Figure: The SecureUML Metamodel

SecureUML

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

Modeling Access Control with SecureUML

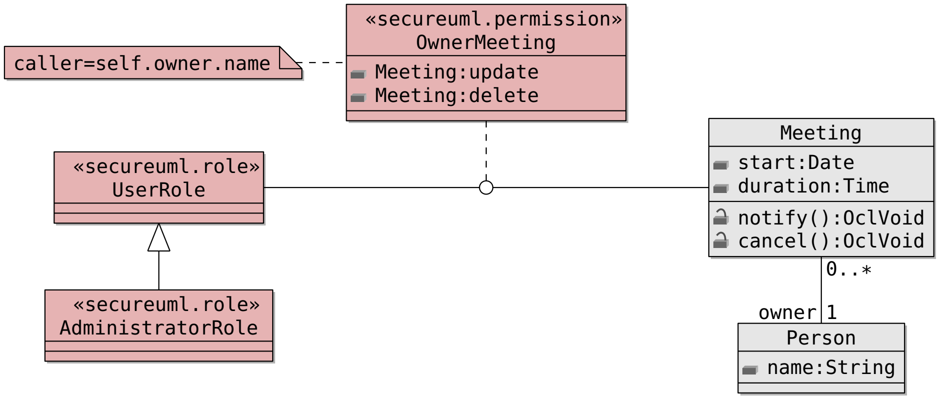
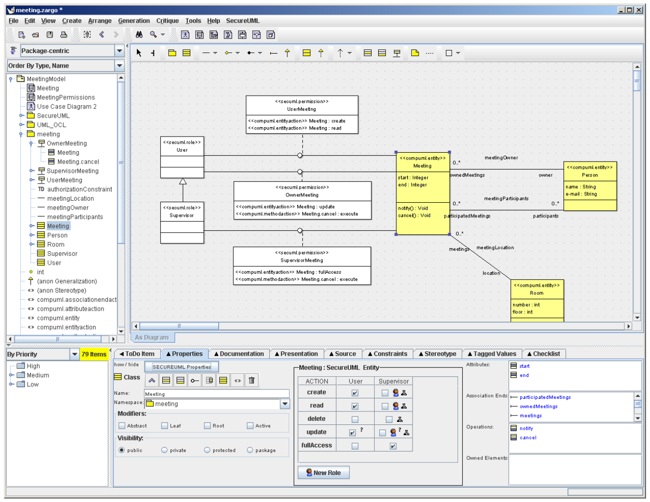


Figure: Access Control Policy for Class Meeting Using SecureUML

Supporting SecureUML in ArgoUML



Name: Meeting

Namespace: meeting

Modifiers: ☐ Abstract ☐ Leaf ☐ Root ☐ Active

Visibility: ● public ○ private ○ protected ○ package

ACTION	User	Supervisor
create	☑	☐
read	☑	☐
delete	☐	☐
update	☑ ?	☐ ?
fullAccess	☐	☑

From SecureUML to UML/OCL

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

The transformation basically

- initializes a concrete authorization environment,
- transforms the design model, and
- transforms the security model.

The Authorization Environment

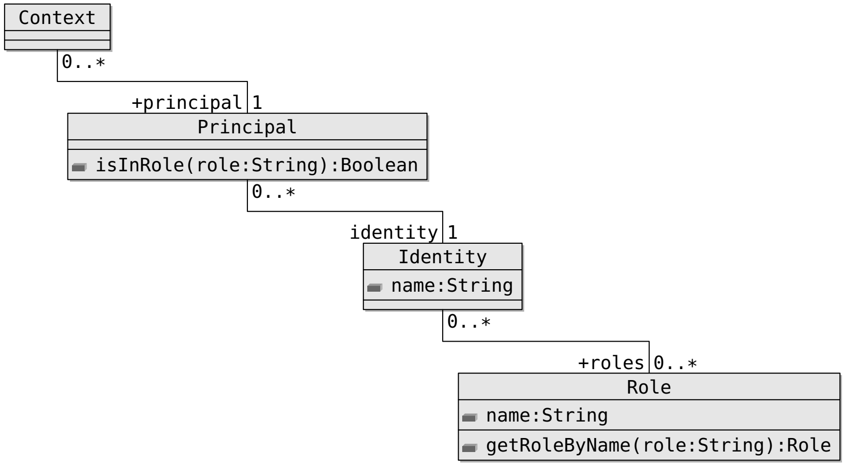


Figure: Basic Authorization Environment

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:` $\overline{\text{post}_{op}} = \text{post}_{op}$`[f() ↦ f_sec(), att ↦ getAtt()]`

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:` $\overline{\text{post}_{op}} = \text{post}_{op}$`[f()` $\mapsto$ `f_sec(), att` $\mapsto$ `getAtt()]`

Security Model Transformation

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

$$\begin{array}{lll} \text{inv}_C & \mapsto & \text{inv}_C \\ \text{pre}_{op} & \mapsto & \text{pre}_{op} \\ \text{post}_{op} & \mapsto & \texttt{if auth}_{op} \\ & & \texttt{then } \overline{\text{post}_{op}} \\ & & \texttt{else result.oclIsUndefined()} \\ & & \qquad \texttt{and Set\{\}->modifiedOnly()} \\ & & \texttt{endif} \end{array}$$

where auth_{op} represents the authorization requirements.

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>')
```

Relative Consistency

- An invariant (class) is **invariant-consistent**, if a satisfying state exists:

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

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

$$\exists \sigma.\ \sigma \models \mathit{inv}_1 \ \text{ and } \ \mathit{inv}_2 \ \text{ and } \ \cdots \ \text{ and } \ \mathit{inv}_n$$

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

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

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*:

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

- 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.