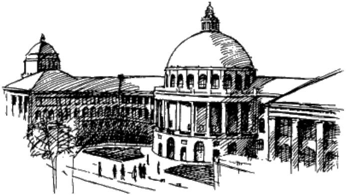
An Interactive Proof Environment for Object-oriented Specifications

Achim D. Brucker
brucker@inf.ethz.ch http://www.brucker.ch/



Information Security, ETH Zurich, Switzerland

March 9th, 2007

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.

We Address Some of These Criticisms

We formalize UML/OCL and provide tool support

- Our solution is formal
- Our solution is based on a standard widely used in industry
- Our solution has tool support

Contributions

- Theory:
  - A formal semantics for constrained OO data structures
  - An extensible, type-safe representation of object-structures in HOL,
  - A formal semantics for OO constraint languages
  - Proof calculi for a three-valued logic over path expressions
- Practice:
  - A machine checked semantics for OCL 2.0
  - A framework for analyzing OO specifications
  - A datatype package for OO data structures,
  - HOL-OCL, an interactive theorem prover for UML/OCL

Contributions

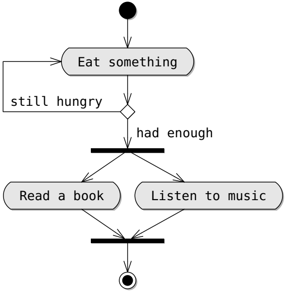
- Theory:
  - An extensible, type-safe representation of object-structures in HOL,
- Practice:
  - A machine checked semantics for OCL 2.0

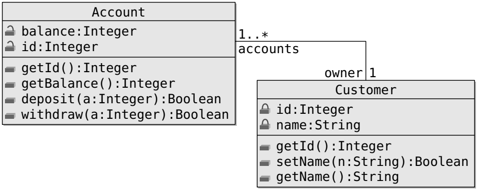
Outline

1. UML/OCL in a Nutshell
   - The Unified Modeling Language (UML)
   - The Object Constraint Language (OCL)
2. Formalization of UML and OCL
   - Formalization of UML
   - Formalization of OCL
3. Conclusions and Outlook
   - Contributions
   - Conclusions
   - Outlook

The Unified Modeling Language (UML)

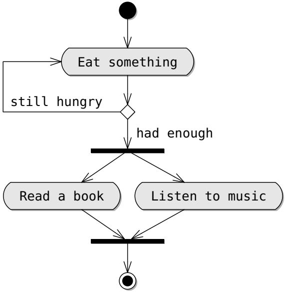
- Visual modeling language
- Object-oriented development
- Industrial tool support
- OMG standard
- Many diagram types, e. g.
    - activity diagrams
    - class diagrams
    - …

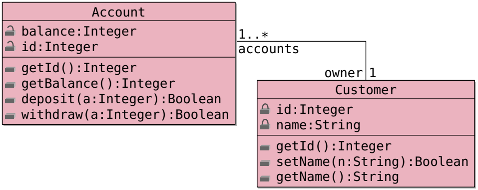




The Unified Modeling Language (UML)

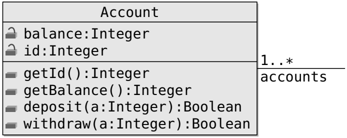
- Visual modeling language
- Object-oriented development
- Industrial tool support
- OMG standard
- Many diagram types, e. g.
  - activity diagrams
  - class diagrams
  - …





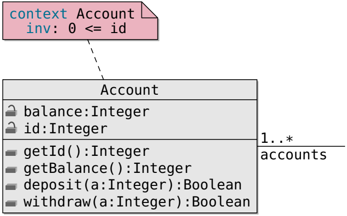
The Object Constraint Language (OCL)

- 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



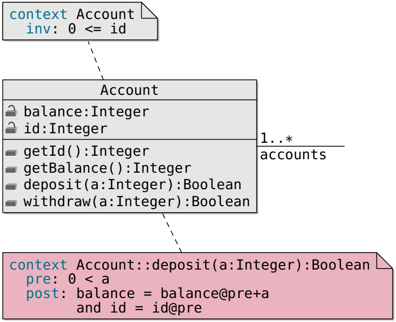
The Object Constraint Language (OCL)

- 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



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

How to Proceed?

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. UML/OCL in a Nutshell
2. Formalization of UML and OCL
   - Formalization of UML
   - Formalization of OCL
3. Conclusions and Outlook

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`.

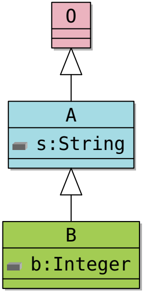
- Formalizing type safe path expressions requires
  - a HOL representation of class types
  - HOL functions for accessing attributes
  - support for inheritance and subtyping
- After adding new classes to a model
  - there is no need for re-proving
  - definitions can be re-used
- Goal: a type-safe object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

Representing Class Types

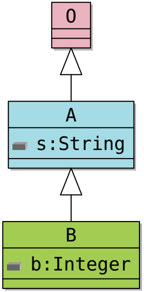
- The "extensible records" approach
  - We assume a common superclass (`O`).
  - The uniqueness is guaranteed by a *tag type*, e. g.:

$$O_{tag} := \mathrm{classO}$$

  - Construct class type as tuple along inheritance hierarchy



Representing Class Types



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

$$O_{tag} := classO$$

  - Construct class type as tuple along inheritance hierarchy

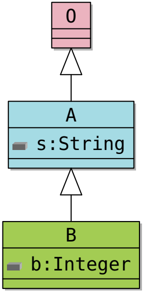
B :=

Representing Class Types

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

$$O_{tag} := classO$$

  - Construct class type as tuple along inheritance hierarchy



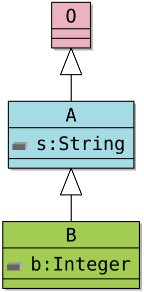
$$B := (O_{tag} \times oid)$$

Representing Class Types

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

$$O_{tag} := \text{classO}$$

  - Construct class type as tuple along inheritance hierarchy



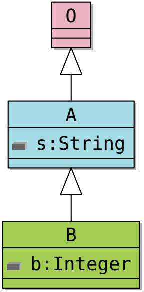
$$B := (O_{tag} \times \text{oid}) \times \Big((A_{tag} \times \texttt{String}) \qquad \Big)$$

Representing Class Types

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

$$O_{\text{tag}} := \text{classO}$$

  - Construct class type as tuple along inheritance hierarchy



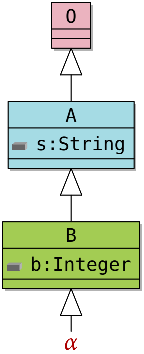
$$B := (O_{\text{tag}} \times \text{oid}) \times \Big((A_{\text{tag}} \times \texttt{String}) \times \big((B_{\text{tag}} \times \texttt{Integer}) \quad \big) \Big)$$

Representing Class Types

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

$$O_{tag} := classO$$

  - Construct class type as tuple along inheritance hierarchy



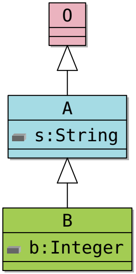
$$\alpha\ B := (O_{tag} \times oid) \times \Big((A_{tag} \times \mathtt{String}) \times \big((B_{tag} \times \mathtt{Integer}) \times \alpha_{\perp}\big)_{\perp}\Big)_{\perp}$$

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

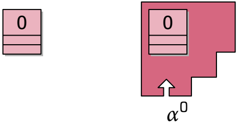


A Universe Type

A universe type represents all classes

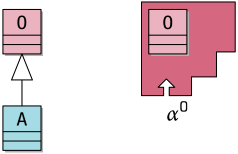
- supports modular proofs with arbitrary extensions
- provides a formalization of a extensible typed object store

An Extensible Object Store



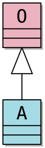
$$\mathscr{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

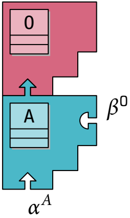
An Extensible Object Store



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

An Extensible Object Store

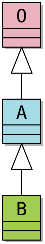


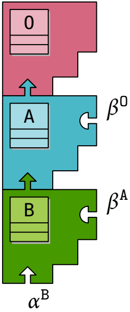


$$\mathscr{U}^{0}_{(\alpha^{O})} = O \times \alpha^{O}_{\bot}$$

$$\mathscr{U}^{1}_{(\alpha^{A},\beta^{O})} = O \times \left(A \times \alpha^{A}_{\bot} + \beta^{O}\right)_{\bot}$$

An Extensible Object Store



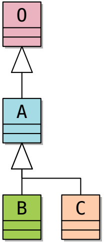


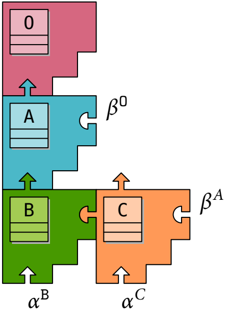
$$\mathcal{U}^0_{(\alpha^{\mathsf{O}})} = O \times \alpha^{\mathsf{O}}_\bot$$

$$\mathcal{U}^1_{(\alpha^A,\beta^{\mathsf{O}})} = O \times \left(A \times \alpha^A_\bot + \beta^{\mathsf{O}}\right)_\bot$$

$$\mathcal{U}^2_{(\alpha^B,\beta^{\mathsf{O}},\beta^A)} = O \times \left(A \times \left(B \times \alpha^B_\bot + \beta^A\right)_\bot + \beta^{\mathsf{O}}\right)_\bot$$

An Extensible Object Store



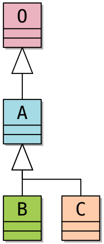


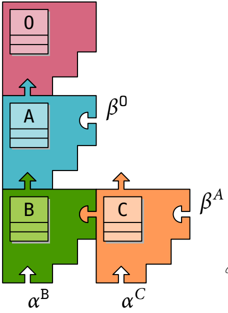
$$\mathcal{U}^0_{(\alpha^{\mathtt{O}})} = O \times \alpha^{\mathtt{O}}_\perp$$

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

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

An Extensible Object Store





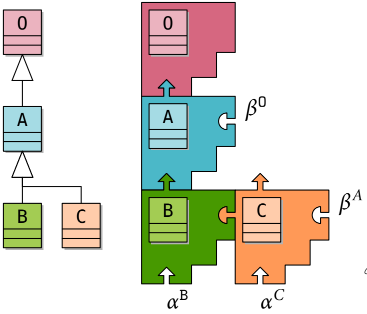
$$\mathcal{U}^0_{(\alpha^0)} = O \times \alpha^0_\perp$$

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

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

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

An Extensible Object Store



$$\mathscr{U}^0_{(\alpha^0)} = O \times \alpha^0_\bot$$

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

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

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

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

Operations Accessing the Object Store

- injections

$$\mathrm{mk}_{\mathrm{O}}\, o = \mathrm{Inl}\, o \qquad \text{with type } \alpha^{O}\ \mathtt{O} \to \mathscr{U}^{0}_{\alpha^{O}}$$

- projections

$$\mathrm{get}_{\mathrm{O}}\, u = u \qquad \text{with type } \mathscr{U}^{0}_{\alpha^{O}} \to \alpha^{O}\ \mathtt{O}$$

- type casts

$$\mathrm{A}_{[\mathrm{O}]} = \mathrm{get}_{\mathrm{O}} \circ \mathrm{mk}_{\mathrm{A}} \qquad \text{with type } \alpha^{A}\ \mathrm{A} \to (A \times \alpha^{A}_{\perp} + \beta^{O})\ \mathtt{O}$$

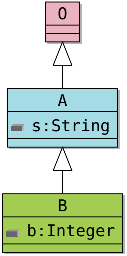
$$\mathrm{O}_{[\mathrm{A}]} = \mathrm{get}_{\mathrm{A}} \circ \mathrm{mk}_{\mathrm{O}} \qquad \text{with type } (A \times \alpha^{A}_{\perp} + \beta^{O})\ \mathtt{O} \to \alpha^{A}\ \mathrm{A}$$

- …

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:

$$\frac{\text{isType}_{\text{B}}\ \mathit{self}}{\text{isKind}_{\text{A}}\ \mathit{self}}$$

- "re-casting":

$$\frac{\text{isType}_{\text{B}}\ \mathit{self}}{\mathit{self}_{[\text{A}][\text{B}]} \neq \bot \wedge \text{isType}_{\text{B}}\ (\mathit{self}_{[\text{A}][\text{B}][\text{A}]})}$$

- monotonicity of invariants, …

All rules are derived automatically

Outline

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:

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

  - a combinator for strictification:

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

Is This Semantics Compliant?

- We prove formally (within our embedding):

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

```
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() $\doteq$ 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

1. UML/OCL in a Nutshell
2. Formalization of UML and OCL
3. Conclusions and Outlook

Contributions

- Theory:
  - A formal semantics for constrained OO data structures
  - An extensible, type-safe representation of object-structures in HOL,
  - A formal semantics for OO constraint languages
  - Proof calculi for a three-valued logic over path expressions
- Practice:
  - A machine checked semantics for OCL 2.0
  - A framework for analyzing OO specifications
  - A datatype package for OO data structures,
  - HOL-OCL, an interactive theorem prover for UML/OCL

Conclusions

- It is possible to formalize a real-world standard
- A shallow embedding can be used for defining the semantics of an object-oriented specification language
- Defining the semantics and building tools conservatively is feasible
- A datatype package for object-oriented data structures is feasible

Outlook

Our framework provides the foundation for:

- Consistency analysis of specifications
- Proving refinement
- Proving side-conditions of model-transformations
- Program verification
- Test data generation

Bibliography

- The Isabelle/HOL-OCL website, Mar. 2006.
- UML 2.0 OCL specification, Oct. 2003.
  Available as OMG document ptc/03-10-14.
- OMG Unified Modeling Language Specification, Mar. 2003.
  (Version 1.5). Available as OMG document formal/03-03-01.
- M. Richters.
  A Precise Approach to Validating UML Models and OCL Constraints.
  PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

Part II

Appendix

Outline

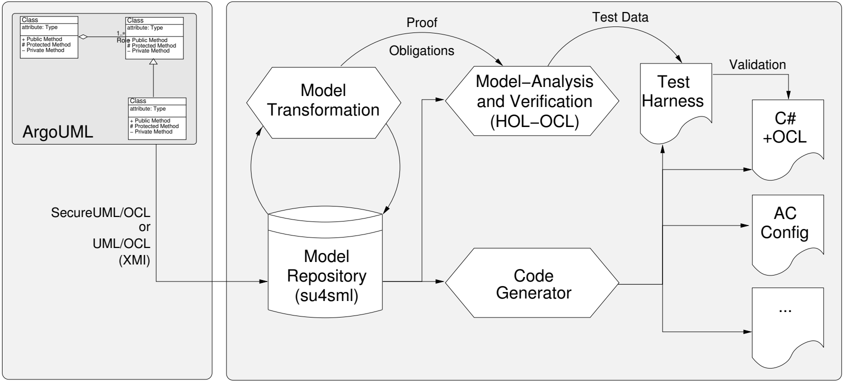
Challenges of Formalizing UML/OCL

Only few formal methods are specialized for analyzing object oriented specifications.

- Problems and open questions:
  - object equality and aliasing
  - embedding of object structures into logics
  - referencing and de-referencing, including “null” references
  - dynamic binding
  - polymorphism
  - representing object-oriented concepts inside λ-calculi
  - providing a (suitable, shallow) representation in theorem provers
  - …

Our Vision:

A Tool-supported Formal Software Development Process



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.

Why use Formal Methods

There are many reasons for using formal methods:

- safety critical applications, e.g. flight or railway control.
- security critical applications, e.g. access control.
- financial reasons (e.g. warranty), e.g. embedded devices.
- legal reasons, e.g. certifications.

Many successful applications of formal methods proof their success!

Why Formal Methods are not widely accepted ?

- Only a few formal methods address industrial needs:
  - support for object-oriented modeling and programming.
  - formal tool support (model checkers, theorem provers, …).
  - integration in standard CASE tools and processes.
- Formal methods people and industrial software developer are often speaking different languages.

To tackle these challenges we provide a a formal foundation for (supporting object-orientation) for a industrial accepted specification languages (UML/OCL) [3, 2].

Is OCL an Answer?

- UML/OCL attracts the practitioners:
  - is defined by the OO 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 OO semantics research,
  - offers the chance for bringing formal methods closer to industry.

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

Are diagrams enough to specify OO systems formally?

- *The short answer:*
  - UML diagrams are not powerful enough for supporting formal reasoning over specifications.
- *The long answer:*
  We want to be able to
  - verify (proof) properties
  - refine specifications
- *Thus we need:*
  - a formal extension of UML.

Strong 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

The Semantic Foundation of OCL

The semantics of OCL 2.0 is spread over several places:

Chapter 7 "OCL Language Description" (informative): introduces OCL informally using examples,

Chapter 10 "Semantics Described using UML" (normative): presents an "evaluation" environment,

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

Appendix A "Semantics" (informative): presents a formal semantics (textbook style), based on the work of Richters.

The Semantics Foundation of the Standard

We see the formal foundation of OCL critical:

- no normative formal semantics.
- no consistency and completeness check.
- no proof that the formal semantics satisfies the normative requirements.

Nevertheless, we think the OCL standard ("`ptc/03-10-14`") is mature enough to serve as a basis for a machine-checked semantics and formal tools support.

List of Glitches

- 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)
  - …

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):

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

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.

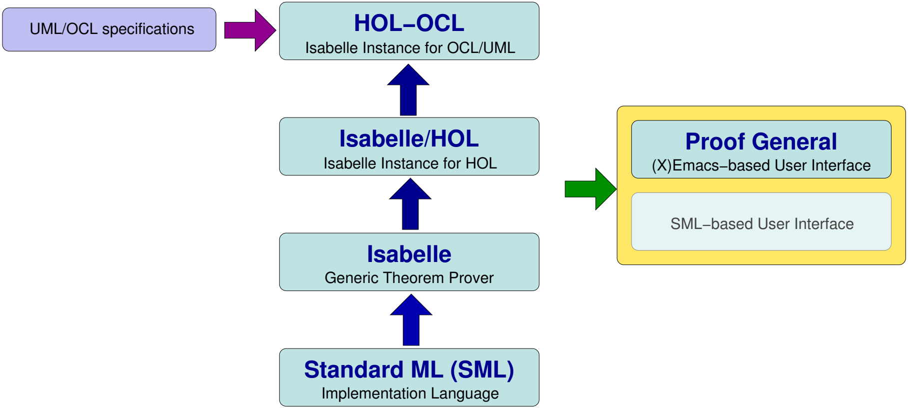
HOL-OCL

- Based on our formalization of UML and OCL, we use Isabelle for developing a “new” theorem prover: HOL-OCL.
- HOL-OCL provides:
  - a formal, machine-checked semantics for OCL 2.0,
  - an interactive proof environment for OCL,
  - servers as a basis for examining extensions of OCL,
  - publicly available:
    `http://www.brucker.ch/projects/hol-ocl/`.

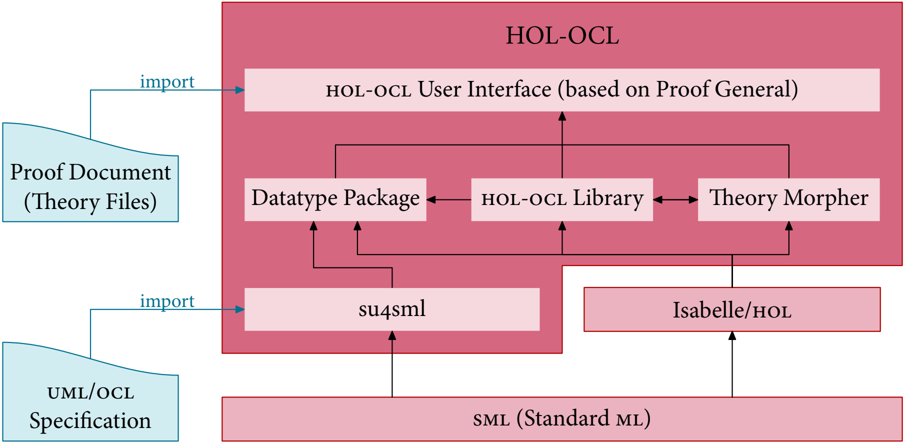
The Technical Design of HOL-OCL

- *Reusability:*
    - Reuse old proofs for class models constructed via inheritance introduction of new classes.
    - Extensible semantics approach.
- *Representing semantics structurally:*
    - Organize semantic definitions by certain combinators capturing the semantical essence (e.g. lifting and strictness).
    - Automatically construct theorems out of uniform definitions.

System Architecture: Overview



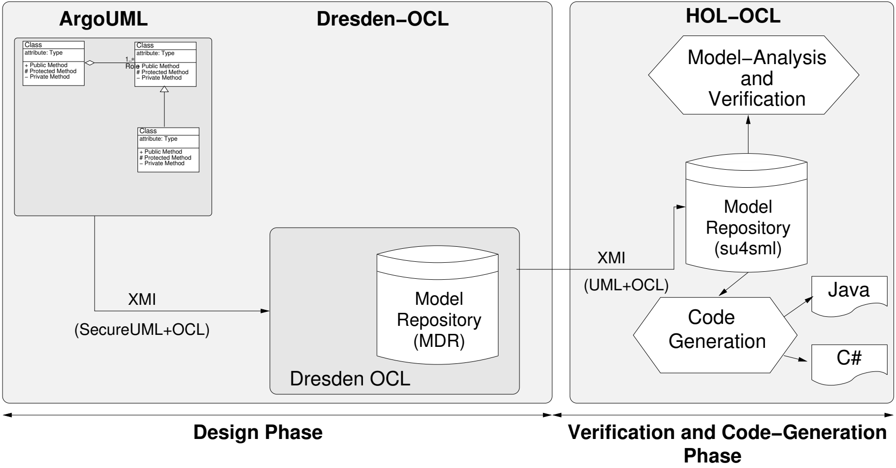
System Architecture: A Detailed View



Programming Isabelle

```
fun cast_class_id class parent thy = let
  val pname = name_of parent
  val cname = name_of class
  val thmname = "cast_"^(cname)^"_id"
  val goal_i = mkGoal_cterm
          (Const(is_class_of class,dummyT)$Free("obj",dummyT))
          (Const("op =",dummyT)$(Const(parent2class_of class pname,dummyT)
            $(Const(class2get_parent class pname,dummyT)$Free("obj",dummyT)))
                                  $(Free("obj",dummyT)))
  val thm = prove_goalw_cterm thy [] goal_i
            (fn p => [cut_facts_tac p 1,  (* proof script *)
                   asm_full_simp_tac
                      (HOL_ss addsimps
                             [o_def,
                              get_def thy (parent2class_of class pname),
                              get_def thy (class2get_parent
                              class pname )]) 1,
                   stac (get_thm thy (Name mk_get_parent)) 1,
                   asm_full_simp_tac (HOL_ss addsimps [
                         get_def thy (is_class_of class),
                         get_thm thy (Name ("is_"^pname^"_mk_"^(cname)))]) 1,
                   stac (get_thm thy (Name ("get_mk_"^(cname)^"_id"))) 1,
                   ALLGOALS(simp_tac (HOL_ss))])
in
  (fst(PureThy.add_thms [((thmname,thm),[])] (thy)))
end
```

The HOL-OCL Workflow



Example 1: Analyzing Redundancies

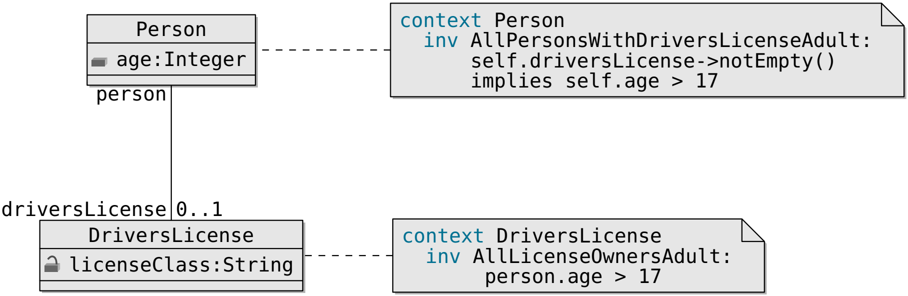


Figure: A simple model of vehicles and licenses

	Invoice	eBank	Company	R&L
classes	3	8	7	13
attributes	5	15	10	27
associations	3	5	6	12
operations	7	8	2	17
generalizations	0	3	0	2
specification (lines)	149	114	210	520
generated theorems	647	1444	1312	2516
time (in seconds)	12	42	49	136

Tools 1/2

	HOL-OCL	KeY	OCLVP
object model	UML	Java	UML
inheritance	single	single	single
extensible	yes	no	no
conservative	yes	no	no
embedding	shallow	pre-compilation	shallow
constraint languge	OCL 2.0	dynamic logic[1]	HOL[2]
conservative	yes	no	yes
invariants	semantic/structural	structural	structural
embedding	shallow	pre-compilation	shallow
datatype package	yes	no	no
meta-logic	HOL	dynamic Logic	HOL

[1] frontend for using OCL 1.x as concrete input syntax available

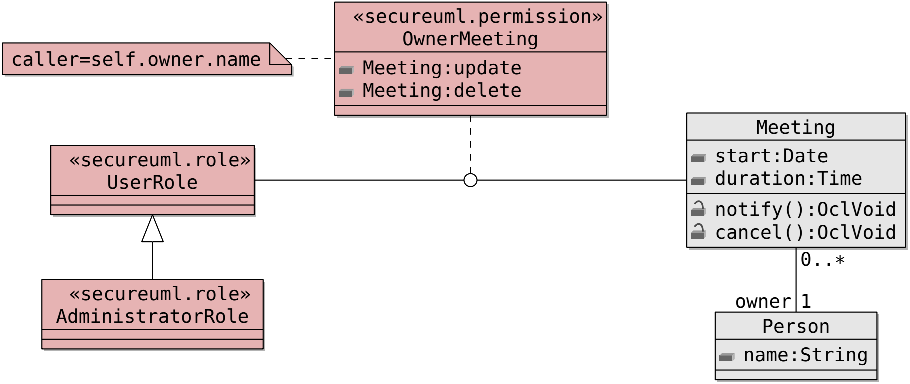
[2] frontend for using OCL 2.x as concrete input syntax available

Tools 2/2

	Boogie	Jive	LOOP
object model	C#	Java	Java
inheritance	single	single	single
extensible	no	no	no
conservative	no	yes	
embedding	pre-compilation	pre-compilation	pre-compilation
constraint languge	Spec#	JML	JML
conservative	no	yes	no
invariants	structural	structral	manual
embedding	pre-compilation	shallow	shallow
datatype package	no	no	no
meta-logic	specialized	HOL	HOL

Future Work: Security (Access Control)

Develop support for analyzing access control in HOL-OCL.



Future Work: Object-oriented Verification

Develop tool-supported, e. g., by extending HOL-OCL, formal methods for object-oriented systems.

Future work in this direction includes

- the development of formal methodologies, e. g., object-oriented refinement.
- the development of methods for source-level verifications.
- the integration of behavioral specifications data-oriented specifications into one consistent formal framework.

Excursus: HOL-TestGen

HOL-TestGenis an formal test-case (test-data) generation tool for the specification-based unit and sequence test.

- Built on top of Isabelle/HOL.
- Test specifications are written in HOL.
- Automatic generation of test scripts.
- Many case-studies: red-black trees, firewall policies, …

```
test_spec "is_sorted(PUT (l::('a list)))"
  apply(gen_test_cases PUT)
store_test_thm "test_sorting"

gen_test_data "test_sorting"
gen_test_script "list_script.sml" test_sorting PUT "myList.sort"
```

Future Work: Specification-based Testing

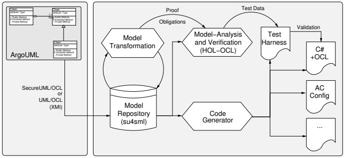
Develop support for analysing access control in HOL-OCL.

Future work in this direction includes

- the integration of (external) automatic decision procedures.
- the development of test-strategies for three-valued specifications.
- The development of domain-specific test-case generation algorithms.

Future Work: Building a Formal (MDE) Toolchain

Building an integrated tool-chain from specification to formal analysis, test-case generation and code-generation.



Future work in this direction includes the development of

- formal model transformation with proof obligations.
- techniques for combining verification and testing.
- techniques for runtime enforcement of specifications.

Personal Motivation and Interests

- My personal interests are centered around:
  - security,
  - formal methods, and
  - software engineering.
- In particular, I want to develop techniques, tools and processes for ensuring
  - correctness,
  - safety, and
  - security

  of software and hardware systems.

Curriculum Vitæ

01/2003–03/2007 Research Assistant at the Information Security Group, headed by Prof. David Basin, ETH Zurich, Switzerland.

06/2000–12/2002 Research Assistant at the Chair for Software Engineering, headed by Prof. David Basin, Albert-Ludwigs University Freiburg, Germany.

06/2000 Diplom Informatiker (Masters of Computer Science), Albert-Ludwigs University Freiburg, Germany. Title of thesis: Verification of Division Circuits using Word-level Decision-diagrams, supervised by Prof. Dr. Bernd Becker.