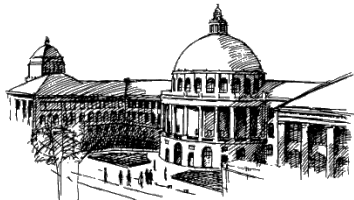


An Interactive Proof Environment for Object-oriented Specifications

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

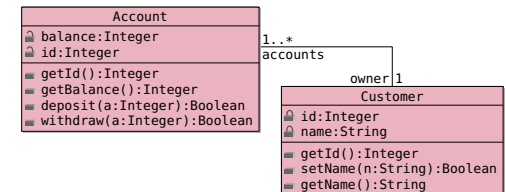
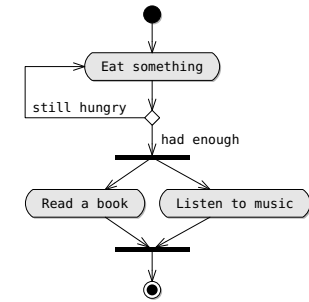
- Theory:
 - A formal semantics for **constrained OO data structures**
 - An **extensible, type-safe** representation of object-structures in HOL,
 - A formal semantics for **OO constraint languages**
 - **Proof calculi** for a three-valued logic over path expressions
- Practice:
 - A **machine checked** 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

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

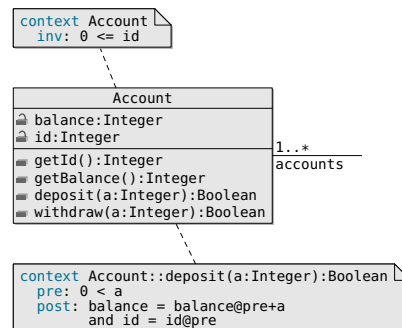
The Unified Modeling Language (UML)

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



The 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
 - ...

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

Outline

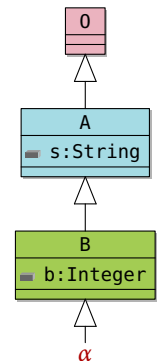
- 1 UML/OCL in a Nutshell
- 2 Formalization of UML and OCL
 - Formalization of UML
 - Formalization of OCL
- 3 Conclusions and Outlook

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

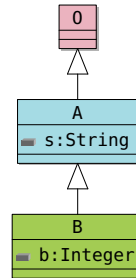


$$\alpha B := (O_{\text{tag}} \times \text{oid}) \times \left((A_{\text{tag}} \times \text{String}) \times \left((B_{\text{tag}} \times \text{Integer}) \times \alpha_{\perp} \right)_{\perp} \right)_{\perp}$$

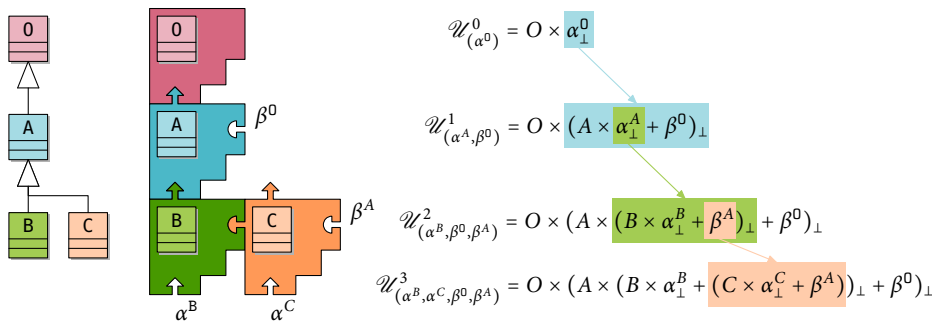
where $__{\perp}$ denotes types supporting undefined values.

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 $\phi(x :: (\alpha \ B))$ 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



An Extensible Object Store



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

A Universe Type

A **universe** type represents all classes

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

Operations Accessing the Object Store

- injections

$$\text{mk}_O o = \text{Inl } o \quad \text{with type } \alpha^O O \rightarrow \mathcal{U}_{\alpha^O}^0$$

- projections

$$\text{get}_O u = u \quad \text{with type } \mathcal{U}_{\alpha^O}^0 \rightarrow \alpha^O O$$

- type casts

$$A_{[O]} = \text{get}_O \circ \text{mk}_A \quad \text{with type } \alpha^A A \rightarrow (A \times \alpha_{\perp}^A + \beta^O) O$$

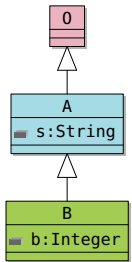
$$O_{[A]} = \text{get}_A \circ \text{mk}_O \quad \text{with type } (A \times \alpha_{\perp}^A + \beta^O) O \rightarrow \alpha^A A$$

- ...

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:

$$\frac{\text{isType}_B \text{ self}}{\text{isKind}_A \text{ self}}$$

- “re-casting”:

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

- monotonicity of invariants, ...

All rules are derived automatically

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)

Outline

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

The OCL Semantics: An Example

- The Interpretation of “X->union(Y)” for sets (“X ∪ 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 “ $X \rightarrow \text{union}(Y)$ ” for sets (“ $X \cup Y$ ”):

$$_ \rightarrow \text{union} _ \equiv \left(\text{strictify}(\lambda X. \text{strictify}(\lambda Y. _ \uparrow X \cup _ \uparrow Y)) \right).$$

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

- a datatype for lifting:

$$\alpha_{\perp} := _ \alpha _ \mid \perp$$

- a combinator for strictification:

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

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

Is This Semantics Compliant?

- We prove formally (within our embedding):

$$\text{Sem}[\text{not } X]y = \begin{cases} _ \neg \text{Sem}[X]y _ & \text{if } \text{Sem}[X]y \neq \perp, \\ \perp & \text{otherwise.} \end{cases}$$

lemma "(Sem[not x]y) = (if Sem[x]y ≠ ⊥ then ⊥¬Sem[x]y else ⊥)"
apply(simp add: OclNot_def DEF_def lifto_def lift1_def lift2_def semfun_def)
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

Outlook





Our framework provides the foundation for:

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

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

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

Challenges of Formalizing UML/OCL

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

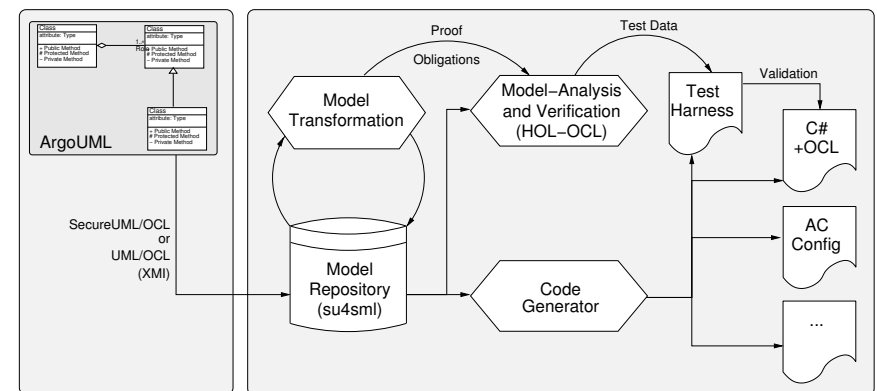
- Problems and open questions:
 - object equality and aliasing
 - embedding of object structures into logics
 - referencing and de-referencing, including “null” references
 - dynamic binding
 - polymorphism
 - representing object-oriented concepts inside λ -calculi
 - providing a (suitable, shallow)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

Outline

- 4 Formalizing UML/OCL
- 5 Motivation
- 6 Background
- 7 Formalizing UML/OCL
- 8 The HOL-OCL System
- 9 Related Work
- 10 Future Work

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 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 formal foundation for (supporting object-orientation) for a industrial accepted specification languages (UML/OCL) [3, 2].

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!

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

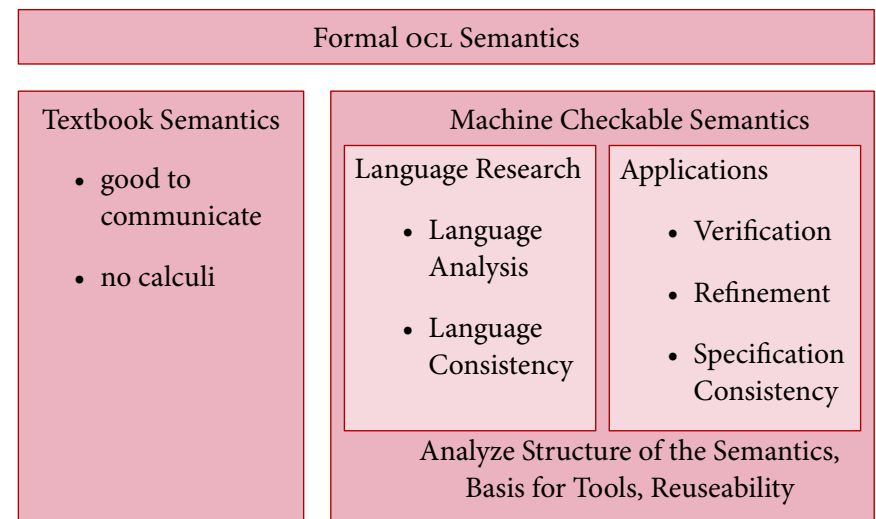
- **shallow embedding:**
Direct definition of the semantics, e.g. each construct is represented by some function on a semantic domain.
 $x \text{ and } y \equiv \lambda e. x e \wedge y e$ $x \text{ or } y \equiv \lambda e. x e \vee y e$

- **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



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.

List of Glitches

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

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.

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: 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 “ $A \rightarrow \text{union } B = B \rightarrow \text{union } A$ ” 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**.

Machine-checked Semantics: Example

Defining the core logic (Strong Kleene Logic):

$$\begin{aligned} \text{not } _ &\equiv \text{lift}_1 \text{strictify}(\lambda x. \lceil \neg x \rceil) \\ _ \text{ and } _ &\equiv \text{lift}_2 (\lambda x y. \text{if } (\text{def } x) \\ &\quad \text{then if } (\text{def } y) \text{ then } \lceil x \wedge y \rceil \\ &\quad \text{else if } \lceil x \rceil \text{ then } \perp \text{ else } \lceil \text{false} \rceil \\ &\quad \text{else if } (\text{def } y) \text{ then if } \lceil y \rceil \text{ then } \perp \\ &\quad \text{else } \lceil \text{false} \rceil \text{ else } \perp) \\ _ \text{ or } _ &\equiv \lambda x y. \text{not } (\text{not } x \text{ and not } y) \\ _ \text{ implies } _ &\equiv \lambda x y. (\text{not } x) \text{ or } y \end{aligned}$$

HOL-OCL

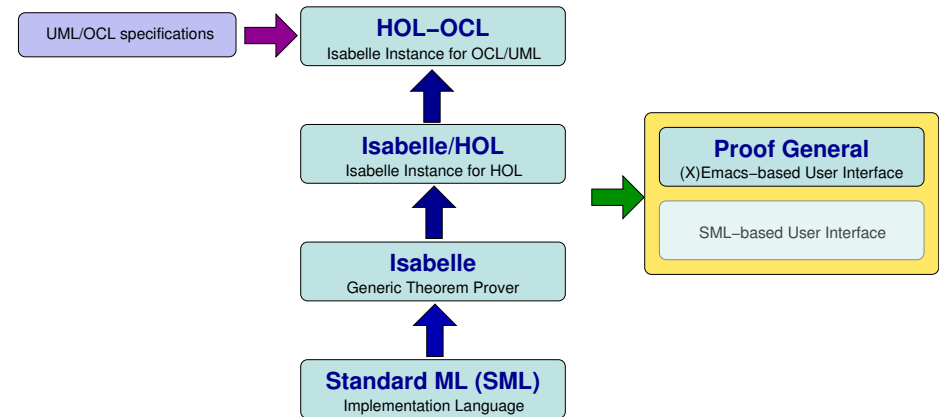


- Based on our formalization of UML and OCL, we use Isabelle for developing a “new” theorem prover: HOL-OCL.
- HOL-OCL provides:
 - a formal, machine-checked semantics for OCL 2.0,
 - an interactive proof environment for OCL,
 - servers as a basis for examining extensions of OCL,
 - publicly available: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

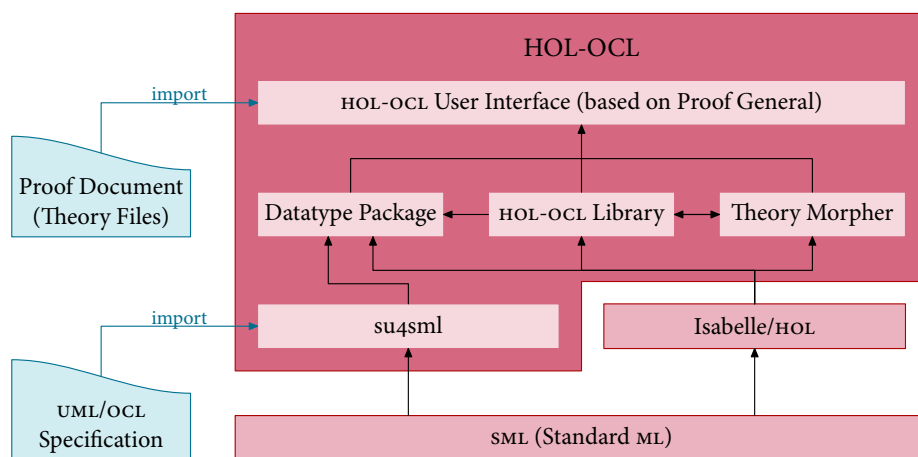
The Technical Design of HOL-OCL

- *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



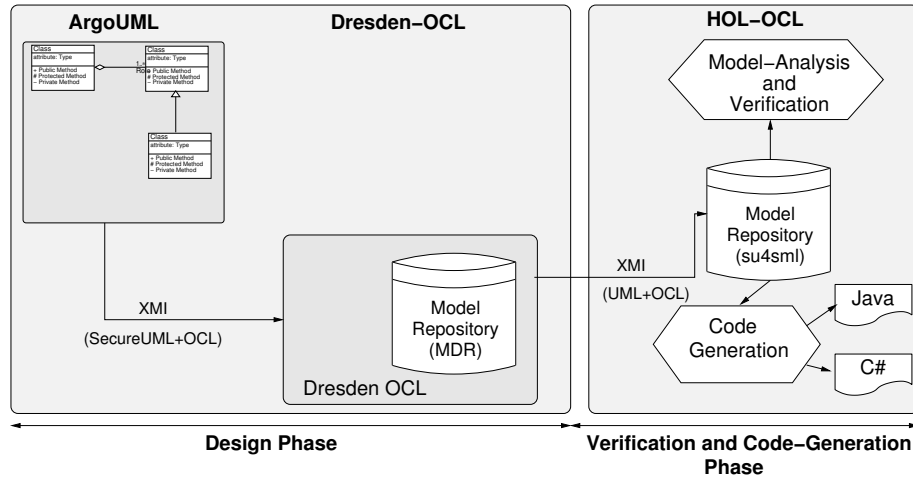
Programming Isabelle

```

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)))
  9   $(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))])
  24 in
  (fst(PureThy.add_thms [((thmname,thm),[]) (thy)])
  end

```

The HOL-OCL Workflow



Example 1: Analyzing Redundancies

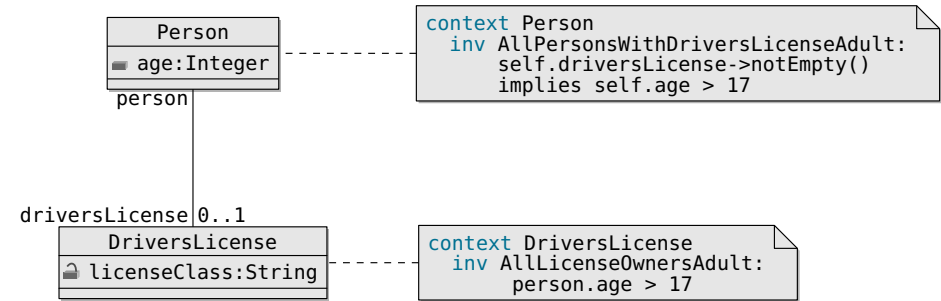


Figure: A simple model of vehicles and licenses

	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 language	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

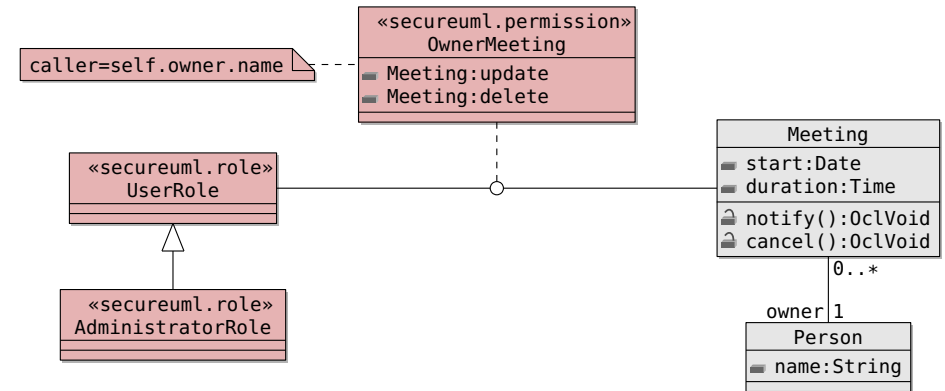
² frontend for using OCL 2.X as concrete input syntax available

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 language	Spec#	JML	JML
conservative	no	yes	no
invariants	structural	structural	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-TESTGEN is 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

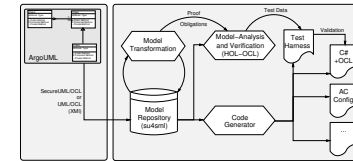
- the integration of (external) automatic decision procedures.
- the development of test-strategies for three-valued specifications.
- The development of domain-specific test-case generation algorithms.

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.

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.

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.