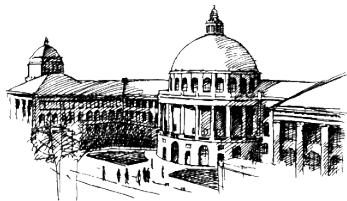


An Application of Isabelle/HOL: HOL-OCL

Achim D. Brucker



Information Security, ETH Zurich, Switzerland

Computer supported Modeling and Reasoning, WS2006
Zurich, January 31, 2007

Outline

Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

Formalizing OCL

HOL-OCL

Conclusions

Bibliography

The Situation Today:

A Software Engineering Problem

- ▶ Software systems
 - ▶ are becoming more and more complex.
 - ▶ used in safety and security critical applications.
- ▶ Formal methods are one way to ensure the correctness.
- ▶ But, formal methods are hardly used by industry.
 - ▶ difficult to understand notation
 - ▶ lack of tool support
 - ▶ high costs
- ▶ Semi-formal methods, especially UML, are
 - ▶ widely used in industry, but
 - ▶ not strong enough for a formal methodologies.

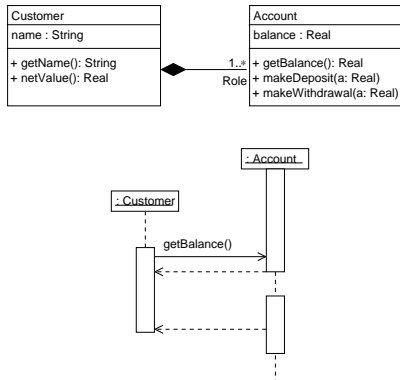
Why Formal Methods are not widely accepted ?

- ▶ 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 a formal foundation for (supporting object-orientation) for a industrial accepted specification languages (UML/OCL) [1, 3].

The Unified Modeling Language (UML)

- ▶ visual modeling language
- ▶ many diagram types, e.g.
 - ▶ class diagrams (static)
 - ▶ state charts (dynamic)
 - ▶ use cases
- ▶ object-oriented development
- ▶ industrial tool support
- ▶ OMG standard with semi-formal semantics

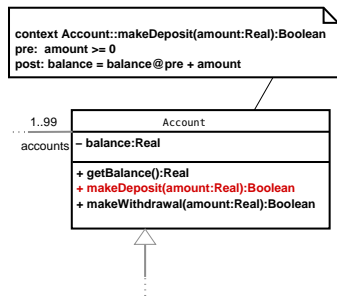


Are UML diagrams enough to specify OO systems formally?

- ▶ *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.

The Object Constraint Language (OCL)

- ▶ extension based on first-order logic with equality and typed set theory
- ▶ designed for annotating UML diagrams
- ▶ in the context of class-diagrams:
 - ▶ preconditions
 - ▶ postconditions
 - ▶ invariants
- ▶ can be used for other diagrams too (not discussed here)



OCL — A Simple Examples

- ▶ “Uniqueness” constraint for the class Account:

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

- ▶ Properties of the class diagram can be described, e.g., multiplicities:

```
context Account inv: Account.owner->size = 1
```

- ▶ Meaning of the method makeDeposit():

```
context Account::makeDeposit(amount:Real):Boolean
pre: amount >= 0
post: balance = balance@pre + amount
```

OCL context

OCL keywords

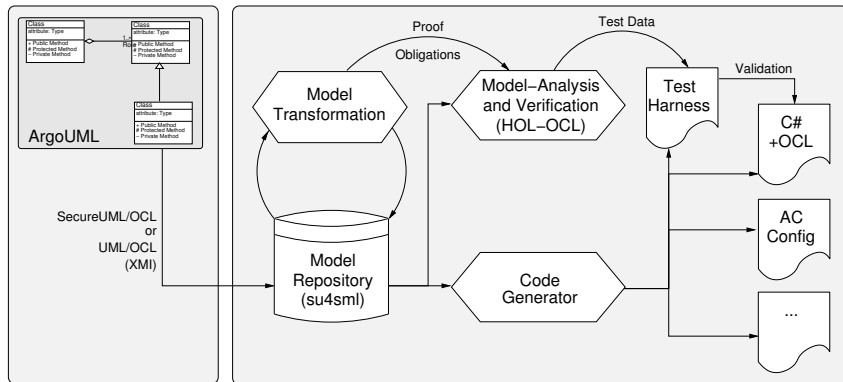
UML path expressions

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.

Our Vision



Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

Formalizing OCL

HOL-OCL

Conclusions

Bibliography

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.

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
 - ▶ ...

How to proceed

For Turning UML/OCL into a formal method we need

1. a **formal semantics** of UML class diagrams.
 - typed path expressions
 - inheritance
 - ...
2. a **formal semantics** of OCL and proof support for OCL.
 - reasoning over UML path expressions
 - large libraries
 - ...

Do the UML and OCL standards provide the needed semantics?

Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

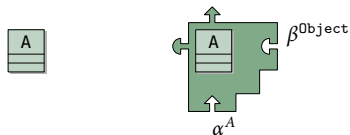
Formalizing OCL

HOL-OCL

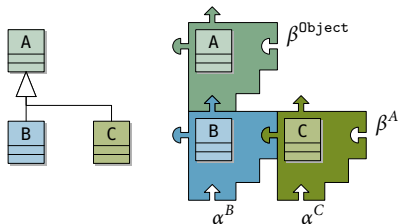
Conclusions

Bibliography

The Object Store



$$\mathcal{U}_{(\alpha^A, \beta^{Object})}^1 = A \times \alpha^A + \beta^{Object}$$



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

Representing Class Types

- ▶ We assume a common superclass (OclAny).
- ▶ Attributes:
 - ▶ basic types (e. g., Integer) are represented directly.
 - ▶ class types are represented by an object identifier (oid).
- ▶ The uniqueness is guaranteed by a special *tag type*.
- ▶ Lifting ($\llcorner _ \lrcorner$) allows for undefined components.
- ▶ Using a type variable allows for extensions (inheritance).

$$\begin{aligned}
 \alpha \text{ Manager} := & ((\text{OclAny}_{\text{tag}}, \text{oid}), \\
 & \llcorner ((\text{Employee}_{\text{tag}}, \text{oid Set, String, Integer}), \\
 & \llcorner ((\text{Manager}_{\text{tag}}, \text{Integer}), \llcorner \alpha \lrcorner) \lrcorner) \lrcorner)
 \end{aligned}$$

Is This Really Modeling Object-orientation?

- ▶ For each UML model, we have to show several properties.
- ▶ For example, for each pair of classes A and B where B inherits from A we derive

$$\frac{\text{self.oclIsType}(B)}{\text{self.oclIsKind}(A)}$$

and

$$\frac{\text{self.oclIsDefined()} \quad \text{self.oclIsType}(B)}{\text{self.oclAsType}(A).\text{oclAsType}(B).\text{oclIsDefined()} \text{ and } \text{self.oclAsType}(A).\text{oclAsType}B.\text{oclIsType}(B)}$$

Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

Formalizing OCL

HOL-OCL

Conclusions

Bibliography

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 1

- ▶ 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 **strict** and **lifted** version of the union of “mathematical sets”.

Textbook Semantics: Example 2

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 1

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

$$_ \rightarrow \text{union} _ \equiv \text{lift}_2 \left(\text{strictify} \left(\lambda X. \text{strictify} \left(\lambda Y. _ \lceil X \rceil \cup _ \lceil Y \rceil \right) \right) \right).$$

- ▶ We make concept like “strict” and “lifted” explicit, i. e.,
 - ▶ Strictifying:

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

- ▶ Datatype for Lifting: $\alpha_{\perp} := _ \lceil \alpha \rceil \mid \text{down}$ and

$$\lceil x \rceil \equiv \begin{cases} v & \text{if } x = _ \lceil v \rceil, \\ \varepsilon \ x. \text{ true} & \text{otherwise.} \end{cases}$$

Machine-checked Semantics: Example 2

Defining the core logic (Strong Kleene Logic):

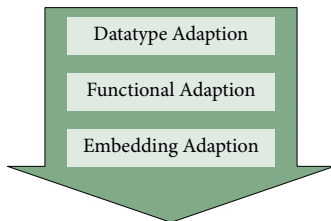
$$\text{not } _ \equiv \text{lift}_1 \text{strictify}(\lambda x. _ \lceil \neg x \rceil _)$$

$$\begin{aligned} _ \text{ and } _ \equiv \text{lift}_2 \left(\lambda x y. \text{if } (\text{def } x) \right. \\ \quad \text{then if } (\text{def } y) \text{ then } _ \lceil x \rceil \wedge \lceil y \rceil _ \\ \quad \quad \text{else if } \lceil x \rceil \text{ then } \perp \text{ else } _ \lceil \text{false} \rceil _ \\ \quad \text{else if } (\text{def } y) \text{ then if } \lceil y \rceil \text{ then } \perp \\ \quad \quad \text{else } _ \lceil \text{false} \rceil _ \text{ else } \perp \left. \right) \end{aligned}$$

$$_ \text{ or } _ \equiv \lambda x y. \text{not } (\text{not } x \text{ and not } y)$$

$$_ \text{ implies } _ \equiv \lambda x y. (\text{not } x) \text{ or } y$$

Meta-language (e.g., HOL)			
<i>Datatype:</i>	bool	int	α' set
<i>Operations:</i>	$\neg, _ \wedge _$	$\neg, _ + _$	$_ \cup _, _ \in _$
<i>Rules:</i>	$x \wedge y = y \wedge x$	$x + y = y + x$	$x \cup y = y \cup x$



Object-language (e.g., OCL)			
<i>Datatype:</i>	Boolean _{τ}	Integer _{τ}	α' Set _{τ}
<i>Operations:</i>	not $\neg, _ \text{ and } _$	$\neg, _ + _$	$_ \rightarrow \text{union } _$
<i>Rules:</i>	$x \text{ and } y = y \text{ and } x$	$x + y = y + x$	$x \rightarrow \text{union } y = y \rightarrow \text{union } x$

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**.

But is This Semantics Compliant?

- ▶ Compliance to the textbook semantics:
 - ▶ We can introduce a semantic mapping

$$\text{Sem}[[x]] \equiv x$$

explicitly and prove formally (within our embedding):

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

- ▶ Compliance to the normative requirements, e. g.:

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

Proving Requirements

isEmpty() : Boolean

(11.7.1-g)

Is self the empty collection?

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

Bag

lemma (self ->isEmpty()) = (self, $\beta :: \text{bot}$)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

Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

Formalizing OCL

HOL-OCL

Conclusions

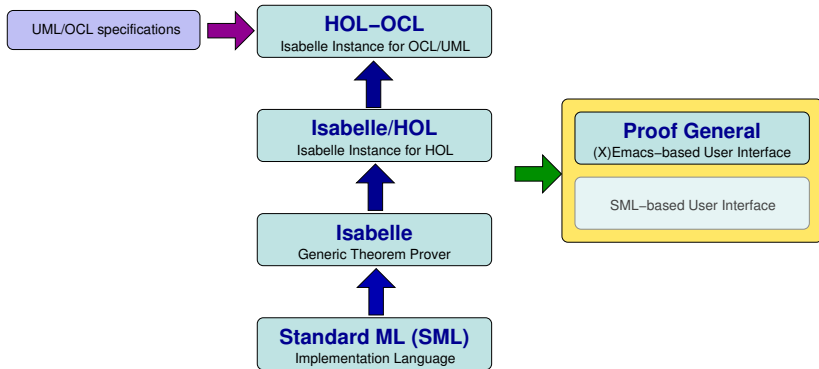
Bibliography

HOL-OCL

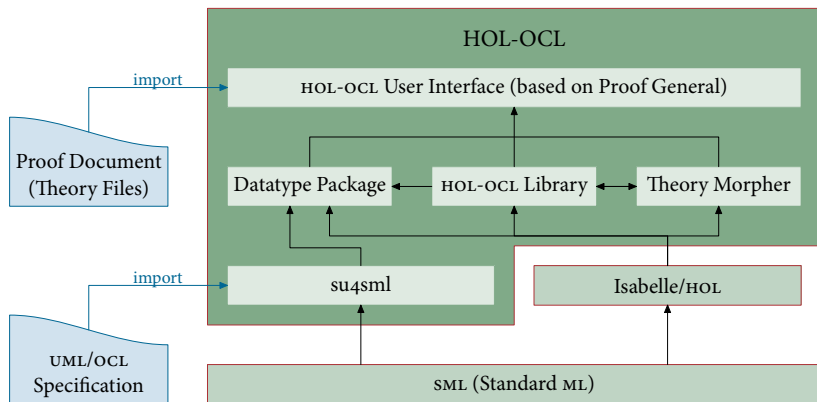


- ▶ Based on our formalization of UML and OCL, we use Isabelle for developing a “new” theorem prover: HOL-OCL.
- ▶ HOL-OCL provides:
 - ▶ a formal, machine-checked semantics for OCL 2.0,
 - ▶ an interactive proof environment for OCL,
 - ▶ servers as a basis for examining extensions of OCL,
 - ▶ publicly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

System Architecture: Overview



System Architecture: A Detailed View



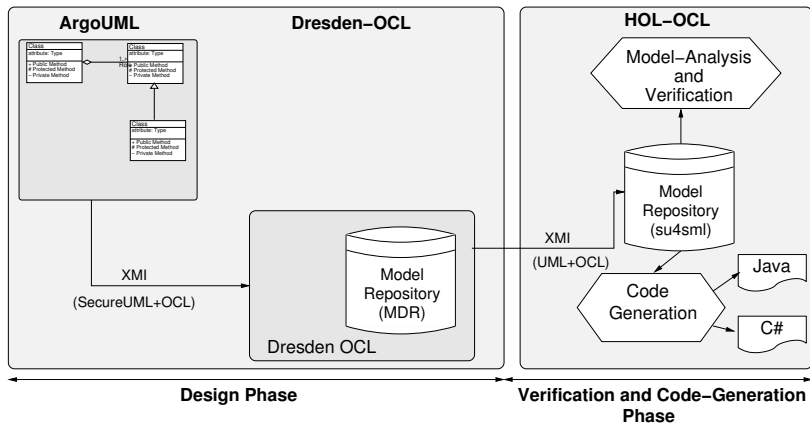
Excursus: Programming Isabelle

```

4 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))])
24 in
  (fst(PureThy.add_thms [((thmname,thm),[])] (thy)))
end

```

The HOL-OCL Workflow



HOL-OCL Demo

Motivation

A Short Introduction to UML/OCL

Turning UML/OCL Into a Strong Formal Method

Formalizing UML

Formalizing OCL

HOL-OCL

Conclusions

Bibliography

What Do We Gain for the OCL community

A machine-checked formal semantics should be a “first class” citizen of the next OCL standard.

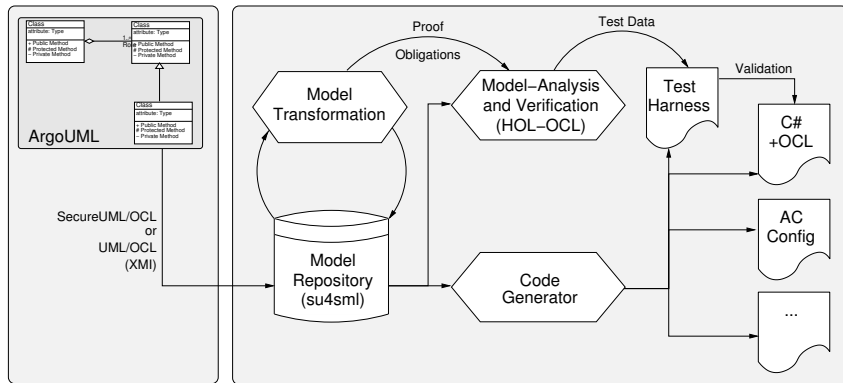
- ▶ UML/OCL could be used for accredited certification process, e. g., Common Criteria,
- ▶ this would open the door for a wide range of semi-formal and formal tools.
- ▶ whereas formalizing too early, can kill the standardization process, for OCL the time is ripe.
- ▶ We provide a formal tool-chain for OCL including code-generators, transformation tools and a theorem prover.





What Do We Show for the Formal Methods People

Formal tools for object-oriented systems can be developed using the conservative, shallow embedding technique.

- ▶ A shallow embedding can be used for defining the semantics of an object-oriented specification language.
- ▶ Defining the semantics, and also building tools, in a conservative way, i. e., without using axioms, is feasible.
- ▶ A conservative embedding technique is useful to compare different semantical variants and possible language extensions.
- ▶ A formalization of a real-world, i. e., defined by an industrial committee, standard of a specification language is possible

Our Vision: Where are we?



-  **OMG Unified Modeling Language Specification, Sept. 2001.**
Available as OMG document formal/01-09-67. This document is superseded by [1].
-  **The Isabelle/HOL-OCL website, Mar. 2006.**
-  **UML 2.0 OCL specification, Oct. 2003.**
Available as OMG document ptc/03-10-14.
-  **M. Richters.**
A Precise Approach to Validating UML Models and OCL Constraints.
PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

Appendix

The OCL Standard

Formal Background

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!

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., 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)
 - ▶ ...

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.

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

▶ **deep embedding:**

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

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

and the explicit semantic function I :

$$I[\text{var } x] = \lambda e. e(x)$$

$$I[x \text{ and } y] = \lambda e. I[x] e \wedge I[y] e$$

$$I[x \text{ or } y] = \lambda e. I[x] e \vee I[y] e$$