HOL-OCL: Experiences, Consequences and Design Choices

Achim D. Brucker and Burkhart Wolff
Albert-Ludwigs Universität Freiburg, Germany

October 3, 2002

This work was partially funded by the OMG member Interactive Objects Software GmbH.

Roadmap

1. Motivation: Use of Semantics
2. Foundations: Isabelle/HOL, HOL-OCL
3. HOL-OCL: Experiences and Applications
4. Conclusion

OCL Semantics

Textbook Semantics
+ Communication
+ Easy to Read
- no Rules
Textbook Semantics: An Example

The interpretation of the logical and is given by a truth-table:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a and b</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>⊥⊥⊥L</td>
<td>⊥⊥⊥L</td>
</tr>
</tbody>
</table>

The Interpretation of “X-union(Y)” for sets (“X ∪ Y”):

\[ I(\cup)(X, Y) \equiv \begin{cases} X \cup Y & \text{if } X \neq ⊥⊥⊥L \text{ and } Y \neq ⊥⊥⊥L \\ ⊥⊥⊥L & \text{otherwise} \end{cases} \]

This is a strict and lifted version of the union of “mathematical sets”.

Textbook Semantics

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

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.
- As basis of further tool support for
  - reasoning over specifications.
  - refinement of specifications.
  - automatic test data generation.

The definition of the logical and (Kleene-logic):

\[
S \land T \equiv \lambda c. \begin{cases} 
\text{if } \text{DEF} (S \ c) \text{ then } \lceil \ [S \ c] \land [T \ c] \rceil \\
\text{else if } S \ c = ([\text{False}]) \text{ then } [\text{False}] \text{ else } \bot \\
\text{else if } T \ c = ([\text{False}]) \text{ then } [\text{False}] \text{ else } \bot 
\end{cases}
\]

The truth-table can be derived from this definition.

The union of sets is defined as the strict and lifted version of \( \cup \):

\[
\text{union} \equiv \text{lift}_2(\text{strictify}_N(\lambda X. \text{strictify}_N( \lambda Y. \text{Abs}_{\text{SSet}} ([\text{Rep}_{\text{SSet}} X] \cup [\text{Rep}_{\text{SSet}} Y] ))))
\]

These definitions can be automatically rewritten into “Textbook-style”.

HOL-OCL: A Shallow Embedding of OCL into HOL

- is build on top of Isabelle/HOL.
- is a shallow embedding of OCL into HOL.
- provides a consistent (machine checked) OCL semantics.
- allows the examination of OCL features.
- builds the basis for OCL tool development.
- follows OCL 1.4 and the RfP for OCL 2.0
- over 2000 theorems (language properties) proven.
The Technical Design of \textit{HOL-OCL}

\textbf{Reuseability:}

– Reuse old proofs for class diagrams constructed via inheritance introduction of new classes.
– Extendible semantics approach.

\textbf{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.

\textit{HOL-OCL} Language Research: Smashed Sets

For handling undefined elements ($\perp$) in Sets we have two possibilities:

1. Not smashed:
   
   \[
   \{X, \perp\} \neq \perp \quad \text{with the consequence} \quad X \in \{X, \perp\} \quad \text{and} \quad \perp \notin \{X, \perp\} 
   \]

2. Smashed:
   
   \[
   \{X, \perp\} = \perp \quad \text{with the consequence} \quad X \not\in \{X, \perp\} \quad \text{and} \quad \perp \notin \{X, \perp\} 
   \]

\textbf{The OCL 2.0 proposal suggest \textit{not smashed} Sets, Bags, Sequences and Tuples:}

\[
I(\text{count} : \text{Set}(t) \times \text{tInteger})(s, v) = \begin{cases} 
1 & \text{if } v \in s \\
0 & \text{if } v \notin s \\
\perp & \text{if } s = \perp 
\end{cases}
\]

And therefore "X->includes(Y)" is \textbf{not executable}!

\textbf{We encourage the use of \textit{smashed} Sets, Bags, Sequences and Tuples:}

– This mirrors the operational behavior of programming languages (e.g. Java).
– This allows the definition of a executable OCL subset.
HOL-OCL: Experiences and Applications

HOL-OCL Application: Test Data Generation

Based on a UML/OCL specification a minimal set of test data is calculated which can be used for validating an implementation.

```
context Triangle : : triangle(s0, s1, s2 : Integer) : TriangType
pre: (s0 > 0) and (s1 > 0) and (s2 > 0)
post: result = if (isTriangle(s0, s1, s2)) then
  if (s0 = s1) then
    if (s1 = s2) then
      Equilateral :: TriangType
    else
      Isosceles :: TriangType
    endif
  else
    Isosceles :: TriangType
  endif
else
  if (s1 = s2) then
    Isosceles :: TriangType
  else
    if (s0 = s2) then
      Isosceles :: TriangType
    else
      Scalene :: TriangType
    endif
  endif
else
  Invalid :: TriangType
endif
```

1. Reduce all logical operation to the basis operators:
   - `and`, `or`, and `not`

2. Determine disjunctive normal Form (DNF):
   - `x and (y or z) ~> (x and y) or (x and z)`

3. Eliminate unsatisfiable sub-formulae, e.g.:
   - `scalene and invalid`

4. Select test data with respect to boundary cases.
Partitioning of the Test Data

triangle $s_0 s_1 s_2 = \neg \text{result} \land \neg \text{isTriangle} s_0 s_1 s_2$

result $\equiv$ invalid and not isTriangle $s_0 s_1 s_2$

or

result $\equiv$ equilateral and isTriangle $s_0 s_1 s_2$ and $s_0 \equiv s_1$ and $s_1 \equiv s_2$

or

result $\equiv$ isosceles and isTriangle $s_0 s_1 s_2$ and $s_0 \equiv s_1$ and $s_1 \neq s_2$

or

result $\equiv$ isosceles and isTriangle $s_0 s_1 s_2$ and $s_0 \equiv s_2$ and $s_0 \neq s_1$

or

result $\equiv$ isosceles and isTriangle $s_0 s_1 s_2$ and $s_1 \equiv s_2$ and $s_1 \neq s_1$

or

result $\equiv$ scalene and isTriangle $s_0 s_1 s_2$ and $s_0 \neq s_1$ and $s_0 \neq s_2$ and $s_1 \neq s_2$

1. Input describes no triangle.

2. Input describes an equilateral triangle.

3. Input describes an isosceles triangle:
   (a) with $s_0$ equals $s_1$.
   (b) with $s_0$ equals $s_2$.
   (c) with $s_1$ equals $s_2$.

4. Input describes a scalene triangle.

For each partition, concrete test data has to be selected with respect to boundary cases (e.g. max./min. Integers, ...).

Conclusion

A theorem prover based OCL definition of the OCL semantics:

- provides a sound and consistent semantic “Textbook”.
- allows the definition of a proof calculi over OCL.
- Gives OCL/UML the power of well-known Formal Methods (e.g. Z, VDM), e.g. for:
  - validation..
  - verification.
  - Refinement.
  - automated test data generation.
  - ...

Conclusion: Tabular overview

<table>
<thead>
<tr>
<th>extendible universes</th>
<th>OCL 1.4</th>
<th>OCL 2.0 RfP</th>
<th>HOL-OCL preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>general recursion</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>smashing</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>automated flattening</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>tuples</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>finite state</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>general Quantifiers</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>allInstances finite</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Kleene logic</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>strong and weak equality</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>
The Unified Modeling Language (UML)

- diagrammatic OO modeling language
- many diagram types, e.g.
  - class diagrams (static)
  - state charts (dynamic)
  - use cases
- semantics currently standardized by the OMG
- we expect wide use in SE-Tools (ArgoUML, Rational Rose, ...)

The Object Constraint Language (OCL)

- designed for annotating UML diagrams (and give foundation for injectivities, ...)
- based on logic and set theory
- in the context of class-diagnostics:
  - preconditions
  - postconditions
  - invariants
- will be used for other diagram types too

Recursive Methods

OCL allows recursive method invocation “as long as the recursion is not infinite”.

For handling non-terminating recursion two possibilities are possible:

- It is forbidden:
  - non-termination is undecidable
  - needs a notion of well-formedness
  - not machine-checkable
  - alternative: well-founded recursion (requires new syntactic and semantic concepts)

- It is undefined (∐): consistent with least-fixpoint in the cpo-theory
Recursive Methods

- We encourage the use of recursive methods, because
  - they are executable
  - increase the expressive power of OCL

- But recursion comes not for free:
  - the semantics of method invocations needs to be clarified.
  - more complexity for code generation tools.

Invariants in OCL

An OCL expression is an invariant of the type and must be true for all instances of that type at any time.

- No problem, as we understand at any time as at any reachable state.
- Intermediate states violating this conditions have to be solved in the refinement notion.
- This also works with general recursion based on fix-points for query-functions.

On Executability of OCL

- The view of OCL as an object-oriented assertion language led to several restrictions, e.g.
  - allInstances() of basic data types is defined as ⊥.
  - states must be finite.

- Thus OCL is not self-contained.
- These restrictions hinder the definitions of general mathematical functions and theorems.
- We suggest to
  1. omit all these restrictions.
  2. define a executable OCL subset.

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.

- deep embedding:
  The abstract syntax is presented as a datatype and a semantic function I from syntax to semantics.
Shallow vs. Deep Embeddings

Representing the logical operations or and and via a

☛ shallow embedding:

\[ x \land y \equiv \lambda e. x e \land y e \quad \text{and} \quad x \lor y \equiv \lambda e. x e \lor y e \]

☛ deep embedding:

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

\[ \text{expr} = \text{var} \mid \text{expr and expr} \mid \text{expr or expr} \]

and the explicit semantic function \( I \):

\[
I[\text{var} x] = \lambda e. e(x) \\
I[\text{expr and expr}] = \lambda e. I[\text{expr}] e \land I[\text{expr}] e \\
I[\text{expr or expr}] = \lambda e. I[\text{expr}] e \lor I[\text{expr}] e
\]