Shortcomings of the Embedding of OCL into QVT ImperativeOCL

Fabian Büttner Mirco Kuhlmann

University of Bremen, Computer Science Department
{green,mk}@tzi.de

September 30, 2008

Talk Structure

Overview of MOF QVT ImperativeOCL

Problems

Suggested Solution

ImperativeOCL

- QVT defines two ways to express transformations: declarative and operational
- ImperativeOCL is the imperative language for the operational approach
- ImperativeOCL extends OCL by:
  - system state manipulation (object creation, property assignment)
  - scoped variables
  - blocks
  - flow control

ImperativeOCL language elements (flow control)

Calculate values procedurally

$$\textbf{compute}(\ x\ : \text{Integer} = 1)\{y := 1; x := y + 1\}$$

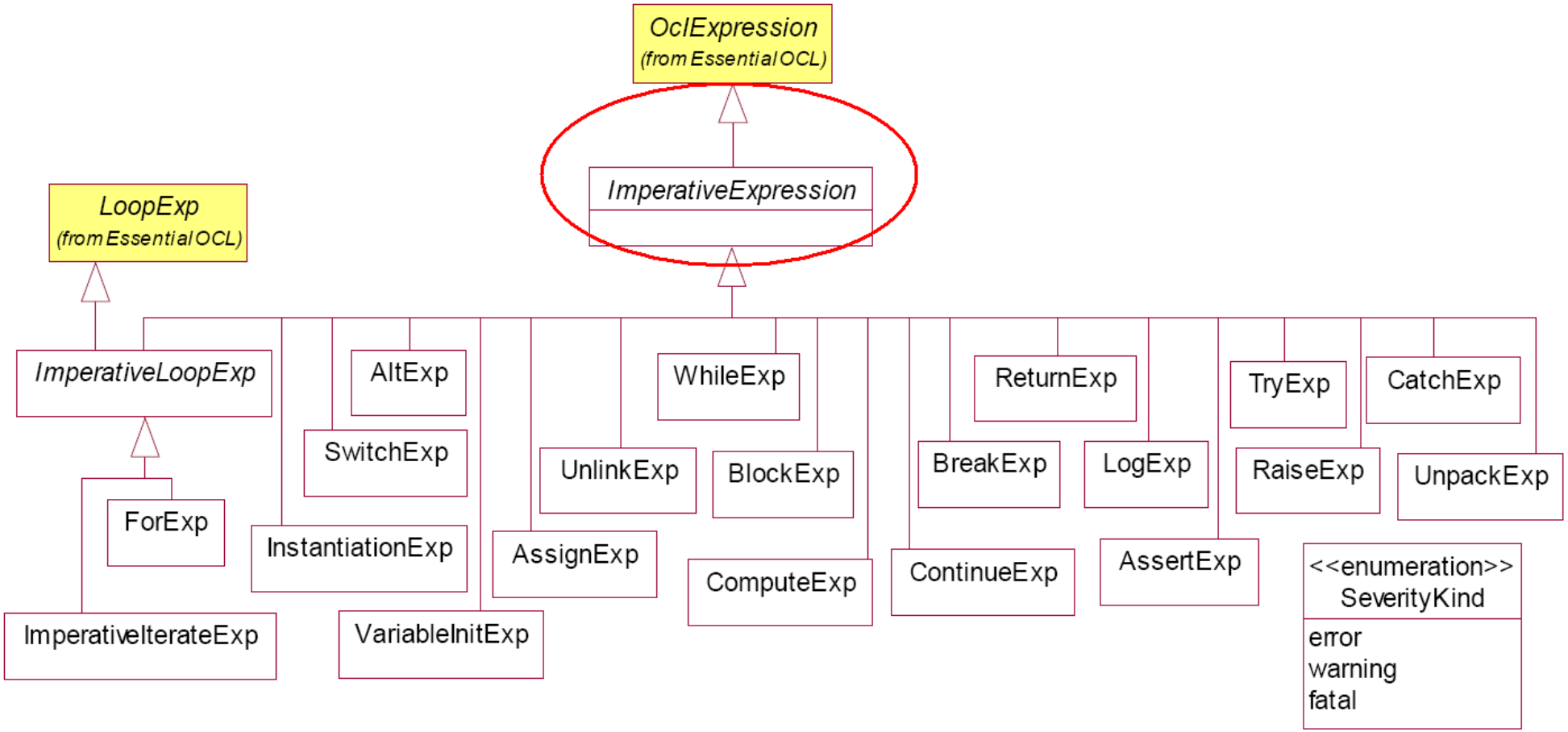
Imperative loops

$$\text{company.employees} \rightarrow \textbf{forEach}(e)\{e.sal := e.sal * 1.1\}$$

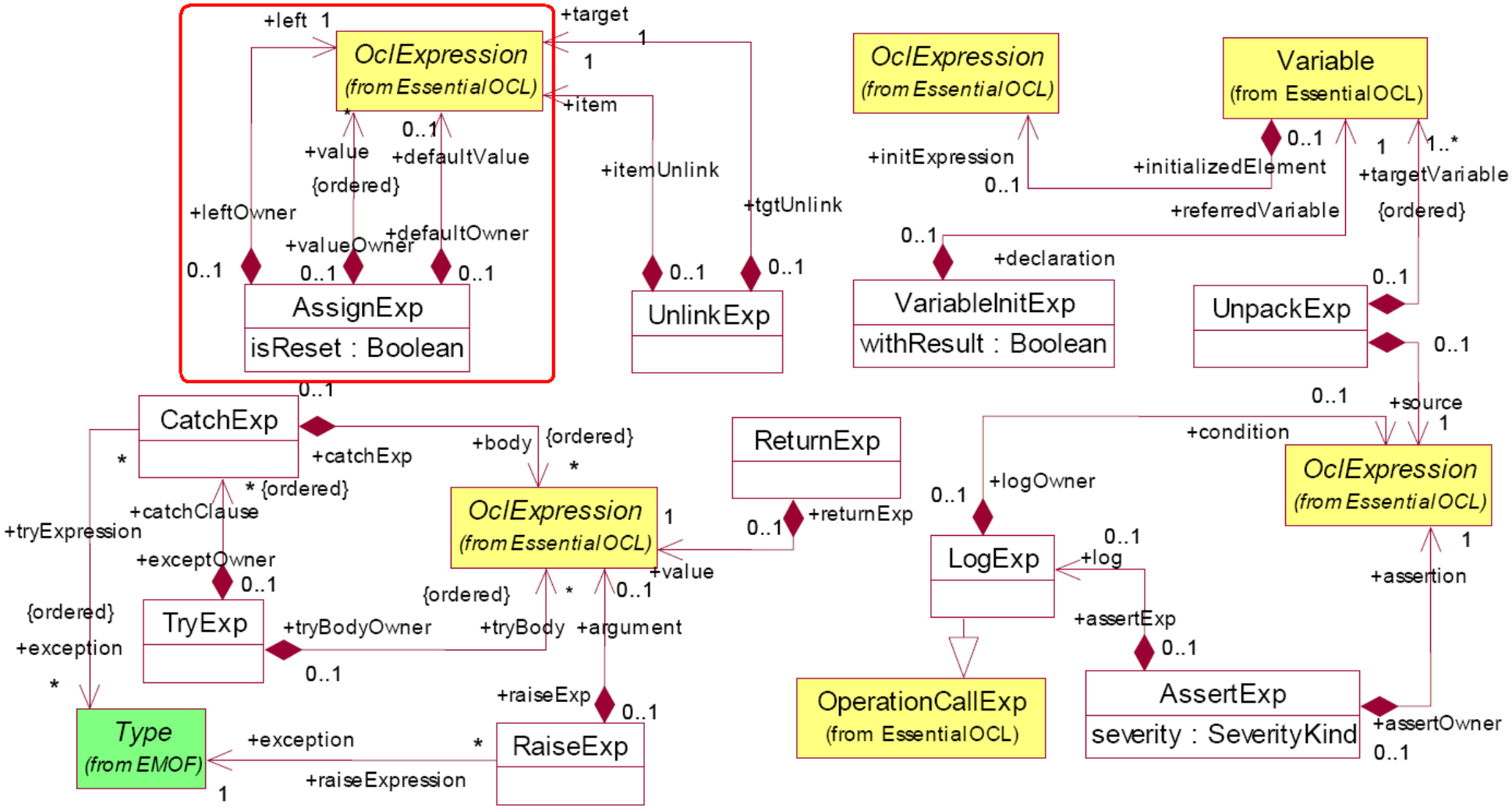
Conditional execution

$$\textbf{if}(x < 0)\ \{x := 0\}\ \textbf{else}\ \{x := 1\}\ \textbf{endif}$$

ImperativeOCL – Abstract Syntax



ImperativeOCL – Abstract Syntax



ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

- An imperative expression

ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

- An imperative expression
- . . . containing an conventional expression . . .

ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

- An imperative expression
- . . . containing an conventional expression . . .
- . . . containing an imperative expression . . .

ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

- An imperative expression
- . . . containing an conventional expression . . .
- . . . containing an imperative expression . . .
- . . . containing an conventional expression

ImperativeOCL expressions $\subset$ OCL expressions

Imperative expressions are defined to be OCL expressions!
Example.

```
z := Set{1,2,3}->select(y |
compute(x:Integer){ x := y * 2 } < 5 )
```

- An imperative expression
- . . . containing an conventional expression . . .
- . . . containing an imperative expression . . .
- . . . containing an conventional expression

This language design creates a lot of problems!

Problem 1 – ImperativeOCL is underdefined

Interpretation for conventional OCL

Given a certain environment (state and variables), the interpretation of an expression is a value:

$$I[\![e]\!] : \mathrm{ENV} \rightarrow \mathrm{VALUE}$$

Problem 1 – ImperativeOCL is underdefined

Interpretation for conventional OCL

Given a certain environment (state and variables), the interpretation of an expression is a value:

$$I[\![e]\!] : \mathrm{ENV} \rightarrow \mathrm{VALUE}$$

Interpretation for ImperativeOCL

The interpretation is a value *and a (modified) environment*:

$$I_{\mathrm{IMP}}[\![e]\!] : \mathrm{ENV} \rightarrow \mathrm{VALUE} \times \mathrm{ENV}$$

Problem 1 – ImperativeOCL is underdefined

Interpretation for conventional OCL

Given a certain environment (state and variables), the interpretation of an expression is a value:

$$I[\![e]\!] : \text{ENV} \rightarrow \text{VALUE}$$

Interpretation for ImperativeOCL

The interpretation is a value *and a (modified) environment*:

$$I_{\text{IMP}}[\![e]\!] : \text{ENV} \rightarrow \text{VALUE} \times \text{ENV}$$

not specified for conventional OCL expressions in MOF QVT!

Problem 1 – ImperativeOCL is underdefined (Example)

```
compute(z:Boolean) {
  var x : Boolean := true
  var y : Boolean := true
  if ((x:=false) and (y:=false)) { ... }
  z := y
}
```

What is the result?

Problem 1 – ImperativeOCL is underdefined (Example)

```
compute(z:Boolean) {
  var x : Boolean := true
  var y : Boolean := true
  if ((x:=false) and (y:=false)) { ... }
  z := y
}
```

What is the result?

Conventional OCL semantics for *and*
(not regarding undefined values for simplicity)

$$I[\![e_1 \text{ and } e_2]\!](\text{env}) := \begin{cases} I[\![e_2]\!](\text{env}) & \text{if } I[\![e_1]\!](\text{env}) = \text{true} \\ \text{false} & \text{otherwise} \end{cases}$$

Problem 1 – ImperativeOCL is underdefined (Example)

```
compute(z:Boolean) {
  var x : Boolean := true
  var y : Boolean := true
  if ((x:=false) and (y:=false)) { ... }
  z := y
}
```

What is the result?

ImperativeOCL – lazy evaluation semantics for (returns true)

$$I_{\mathrm{IMP}}[\![e_1 \text{ and } e_2]\!](\mathrm{env}) := \begin{cases} I_{\mathrm{IMP}}[\![e_2]\!](\mathrm{env}_1) & \text{if } v_1 = \mathrm{true} \\ (\mathrm{false}, \mathrm{env}_1) & \text{otherwise} \end{cases}$$

where $(v_1, \mathrm{env}_1) = I_{\mathrm{IMP}}[\![e_1]\!](\mathrm{env})$.

Problem 1 – ImperativeOCL is underdefined (Example)

```
compute(z:Boolean) {
  var x : Boolean := true
  var y : Boolean := true
  if ((x:=false) and (y:=false)) { ... }
  z := y
}
```

What is the result?

ImperativeOCL – strict evaluation semantics (returns false)

$$I_{\mathrm{IMP}}[\![e_1 \text{ and } e_2]\!](\mathrm{env}) := \begin{cases} (\mathrm{true}, \mathrm{env}_2) & \text{if } v_1 = \mathrm{true} \wedge v_2 = \mathrm{true} \\ (\mathrm{false}, \mathrm{env}_2) & \text{otherwise} \end{cases}$$

where $(v_1, \mathrm{env}_1) = I_{\mathrm{IMP}}[\![e_1]\!](\mathrm{env})$
and $(v_2, \mathrm{env}_2) = I_{\mathrm{IMP}}[\![e_2]\!](\mathrm{env}_1)$.

Problem 2 – Breaks existing equivalence rules

Several equivalence rules we know from logic do not longer hold:

- let $x : T = e_1$ in $e_2 \Leftrightarrow e_2\{x/e_1\}$
- e_1 and $e_2 \Leftrightarrow e_2$ and e_1
- ...

Problem 3 – Redundancy of existing OCL features

Some new features are not really necessary – their effects can be achieved using conventional OCL expressions:

```
company.employees->forEach(c) {
  c.salary := c.salary * 1.1
}
```

has the same effect as

```
company.employees->iterate(c; r:OclAny=Undefined |
  c.salary := c.salary * 1.1
)
```

and

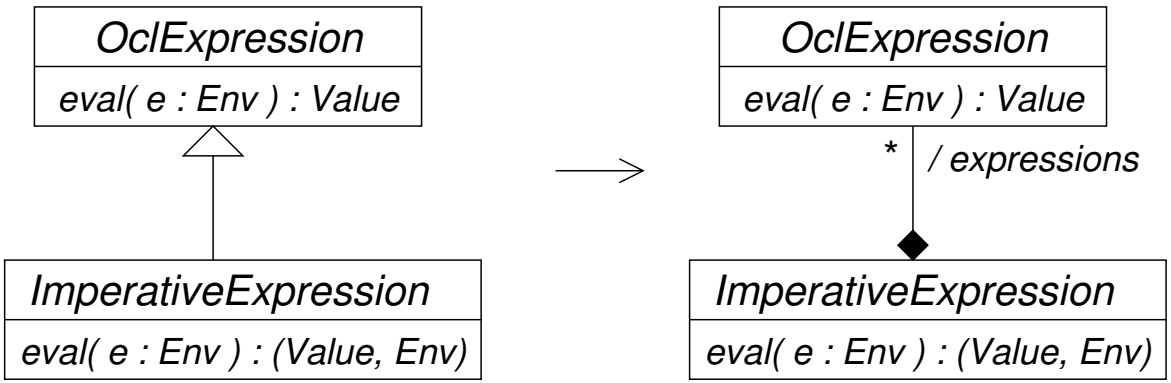
```
if ( x < 0 ) { x := 0 } else { x := 1 } endif
```

is the same as

```
if x < 0 then x := 0 else x := 1 endif
```

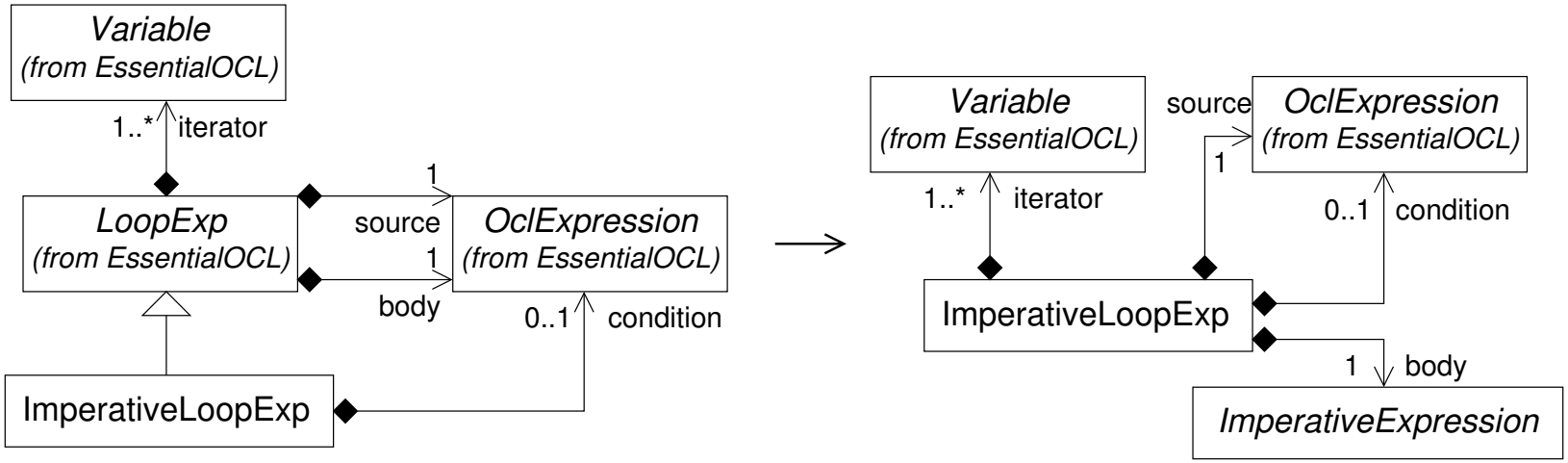
Suggested Solution

Reuse OCL by composition rather than by inheritance in QVTs imperative language



Suggested Solution

Separate imperative statements from side-effect free expressions



Suggested Solution – Tradeoff

Some constellations are not longer possible. While imperative expressions can still contain OCL expressions, OCL expressions can no longer contain imperative expression. Example:

```
z := Set{1,2,3}->select(y |
  compute(x:Integer) { x := y * 2 } < 5
)
```

will be no valid expression. Rework as

1. `z := Set{1,2,3}->select(y | y * 2 < 5)`
2. ```
   z := Set{}
   Set{1,2,3}->forEach(y) {
     if (compute(x:Integer){ x := y * 2 } < 5) { z += y }
   }
   ```
   ```

Thanks for your attention! Questions?