Shortcomings of the Embedding of OCL into QVT ImperativeOCL

Fabian Büttner and Mirco Kuhlmann

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

Abstract. MOF QVT introduces ImperativeOCL as an imperative language for operational descriptions of model transformations (QVT operational mappings). ImperativeOCL extends conventional OCL by expressions with side-effects. A couple of semantical problems arise from the way OCL is embedded into ImperativeOCL – imperative expressions are modelled as a subtype of OCL expressions. This paper points out these semantical problems and proposes a change to the operational mappings language of QVT that resolves these problems, following an approach that reuses OCL by composition rather than by inheritance in the abstract syntax of ImperativeOCL. The proposed change reduces the complexity of the imperative language, removes undefinedness, and leaves OCL conformant to its original definition.

1 Introduction

OCL [OMG06] has proven to be a valuable ingredient in modeling, model validation, and model transformation. It can be used to precisely describe model constraints such as invariants, guards, and pre- and post-conditions, and to formulate queries to system states in general. In model transformation, it can be used to express queries to models, e.g., to specify source objects for transformations.

By now, several OCL tools exist, including ATL [ABJK06], the Dresden OCL toolkit [DOT08], Eclipse MDT OCL [MOT08], KMF [AP05], OCLE [COT08], Octopus [Kla05], RoclET [RT08], and USE [GBR07].

In MOF QVT [OMG08] OCL is extended to so-called ImperativeOCL as part of QVT's "operational mappings". Within ImperativeOCL, statements with side-effects can be formulated. It adds facilities to manipulate system states (e.g, to create and modify objects, links, and variables) and certain constructs from imperative programming languages (e.g., loops, conditional execution). ImperativeOCL is used in QVT to specify transformations operationally (complementary to the relational language of QVT).

While the usefulness of a combination of OCL with imperative language elements is unquestioned, we criticise the way OCL is extended to ImperativeOCL. The chosen abstract syntax leads to a couple of semantical problems which we point out in this paper. In our opinion, the intention of ImperativeOCL can be achieved without modifying the semantics of OCL itself following an approach that favours a composition of OCL into an imperative language over reuse by inheritance in the abstract syntax.

This paper is structured as follows: In Sect. 2 we give a short overview of ImperativeOCL and its role in QVT. In Sect. 3 we explain the various semantical problems that arise from the QVT definition of ImperativeOCL. We suggest a change to the QVT specification that resolves these problems in Sect. 4 and conclude in Sect. 5.

2 ImperativeOCL

QVT defines two ways to express model transformations, a declarative approach and an operational approach. The declarative approach is the Relations language where transformations between models are specified as a set of relations that must hold for the transformation to be successful. The operational approach allows either to define transformations using a complete imperative approach or allows complementing relational transformations with imperative operations implementing the relations. The imperative language introduced by QVT is called ImperativeOCL.

ImperativeOCL adds imperative elements to OCL which are typically found in general purpose programming languages such as Java. Its semantics is defined in [OMG08] as usual by an abstract syntax model. The complete abstract syntax of ImperativeOCL is depicted in Fig. 1.

An element of this extension is the ability to use block expressions to calculate a given value. The compute expression (meta-class ComputeExp in Fig. 1)

```
compute( v:T=initExpr ) { e1; ... ; en }
```

returns the value of the variable *v* after ending the sequential execution of the body *(e1; ... ;en)*. Within the body, variables defined in outer scopes can be freely accessed and changed.

New loop expressions such as *forEach* and *while* have been introduced to iteratively execute expressions (meta-classes ForExp and WhileExpr):

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

while(x<10) { x := x + 2 }
```

An imperative version of conditional evaluation is available (meta-class AltExp):

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

Variables can be declared in the current scope using the *var* statement (meta-class VariableInitExp):

```
var x : String
var y : Integer := 2
```

Instances of classes can be created using a *new* operator (meta-class InstantiationExp):

```
var p : Person
p := Person.new
```

The most important aspect of the abstract syntax to that we refer in this paper is the fact that all imperative expression classes inherit from OclExpression (at the top of Fig. 1). OclExpression is the base class for all expressions in conventional OCL (cf. [OMG06]). Consequently, ImperativeExpressions can be used at all locations where OclExpressions occur. Thus, we can have imperative expressions consisting of OCL expressions that again consist of imperative expressions. For example,

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

becomes a valid OCL expression under the QVT extension. The right-hand side of this imperative assignment expression is an OCL select expression. This OCL select expression requires a boolean body expression, which is given by a relational OCL expression whose left-hand side is an imperative compute expression. The body of this compute expression is an assignment expression whose right-hand side is again a conventional OCL expression (y * 2). While this is a very simple example, trickier mixtures of imperative expressions and OCL expressions exist that comprise several semantical problems. In the following section we explain these problems that all follow from the design by inheritance of ImperativeOCL.

3 Problems

This section explains semantical problems that arise from the embedding of OCL into QVT ImperativeOCL. First (and most formally) we show that ImperativeOCL redefines the interpretation of OCL expressions and that this redefined interpretation leads to undefined semantics of several OCL expressions that had a perfectly well-defined semantics under conventional OCL. Second, we show that several equivalence rules for OCL not longer hold if ImperativeOCL is around. Third, we further show that (under the current design) some of the new imperative expressions are actually redundant to conventional OCL expressions. Finally, we generalise this critique and discuss that the abstract syntax of ImperativeOCL violates the subtype substitution principle in various other locations in UML, too. All of these problems arise from the fact that the abstract syntax of ImperativeOCL allows imperative expressions at all locations where OCL expressions are expected – ImperativeExpression is modeled as a subclass of OclExpression.

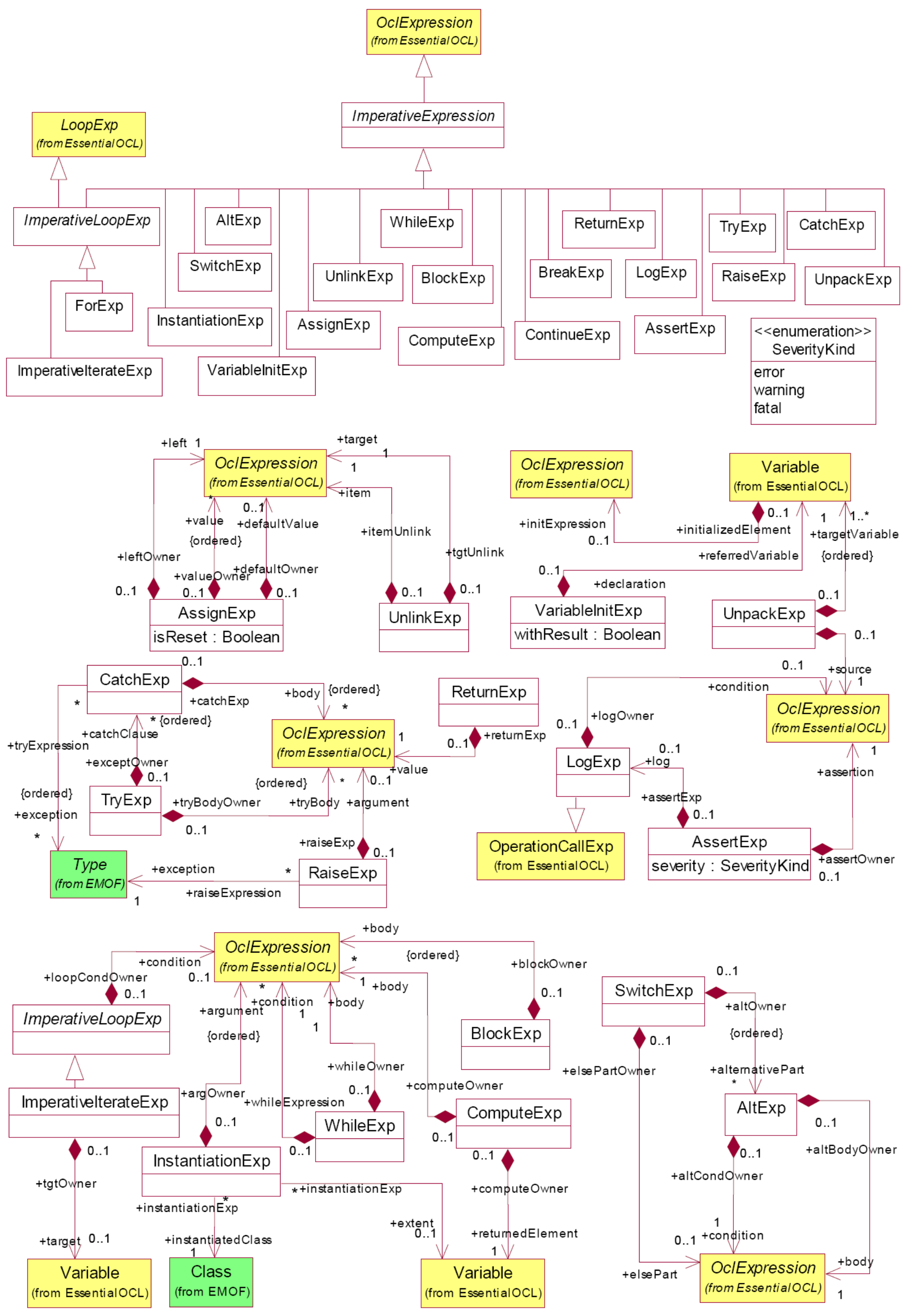

Fig. 1. ImperativeOCL abstract syntax

3.1 Undefined Semantics for OCL Expressions

In conventional OCL, the semantics of an OCL expression e can be formally expressed by an interpretation function

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

where $\mathrm{ENV} = (\sigma, \beta)$ is the environment in which the expression is evaluated (σ is a system state – objects, links, and attribute values –, and β maps bound variables to their values). Cf. [OMG06, Annex A] and [Ric02] for the formal semantics of OCL.

However for ImperativeOCL expressions, the interpretation function described above is not sufficient. The evaluation (or execution) of an ImperativeOCL expression does not only return a value, it also results in a (possibly) modified state and a (possibly) modified variable binding. To take this into account, the interpretation of an ImperativeOCL expression must be defined like follows:

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

While this is not done formally in [OMG08], the effect of all imperative expressions are described in natural language, and one could define $I_{\mathrm{IMP}}[\![e]\!]$ for all imperative expressions from this descriptions.

But, since ImperativeExpression is modeled as a subclass of OclExpression, the imperative semantics MUST be defined for all "ordinary" OCL expression, too, as ImperativeOCL expression can occur everywhere a OCL expression is expected. Figure 2 depicts this structural problem using the OCL metamodel – the redefinition of eval() in ImperativeExpression is invalid.

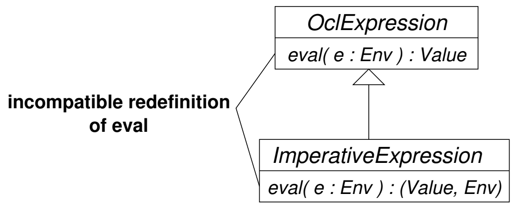
OclExpression
eval(e : Env) : Value
incompatible redefinition of eval
ImperativeExpression
eval(e : Env) : (Value, Env)

Fig. 2. Problem illustrated using the OCL meta-model

The following imperative OCL expressions illustrates this problem:

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

The value of this compute expression is false: the block is executed and the value of the block variable z at the end of the block becomes the value of the compute expression. Since false is assigned to x in the condition of the if statement, z becomes false at the end (the assignment expression has the value of its right-hand side, like in C or Java, therefore the expression x:=false is false).

But what happens if we change the last line as follows:

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

Is the value of this expression true or false? It depends on how we define the imperative semantics of the logical connectives. Given boolean expressions e_1 and e_2, we have at least two choices to define $I_{\text{IMP}}[\![\, e_1 \text{ and } e_2 \,]\!](\text{env})$:

1. Lazy evaluation semantics like in Java or C (returns true for the above example):

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

   where $(v_1, \text{env}_1) = I_{\text{IMP}}[\![\, e_1 \,]\!](\text{env})$. Under this semantics (also called short-circuit evaluation) the right-hand side of the *and* operator is not evaluated if the left-hand side already evaluates to *false*. Therefore, y stays *true*.
2. Strict evaluation semantics (returns false for the above example):

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

   where $(v_1, \text{env}_1) = I_{\text{IMP}}[\![\, e_1 \,]\!](\text{env})$ and $(v_2, \text{env}_2) = I_{\text{IMP}}[\![\, e_2 \,]\!](\text{env}_1)$. Under this semantics, both sides of the *and* operator are always evaluated. Therefore, *false* is assigned to y.

The QVT specification does not say which semantics should hold. But since ImperativeOCL expressions can occur everywhere OCL expressions can occur, this semantics has to be defined.

One can find further similar locations where the imperative semantics of OCL expression is not obvious, e.g. for the collection operation iterate (what happens if the body of the expressions modifies the range variable?) or the treatment of undefined values in arithmetic expressions (similar to the logical connectives – lazy or not?).

3.2 Breaking Equivalence Rules

In conventional OCL, several equivalence rules hold, most of them well-known from predicate logic. If we include imperative expressions into the set of OCL expressions, they all do not longer hold. This is not necessarily a problem but at least contrary to the logical character of conventional OCL.

1. Substituting variables by let expressions. In conventional OCL, the following equivalence holds:

   $$\text{let } x : T = e_1 \text{ in } e_2 \Leftrightarrow e_2\{x/e_1\}$$

   In ImperativeOCL, this equivalence does not hold. The left-hand and right-hand term are only equivalent if x occurs exactly once in e2.
2. Commutativity laws.

   $$e_1 \text{ and } e_2 \Leftrightarrow e_2 \text{ and } e_1$$

   In ImperativeOCL, the commutativity laws for conjunction (and also disjunction) do not longer hold. Notice that this is a different problem than the one discussed in subsection 3.1. The following example illustrates it (returning false and true):

   ```
   compute(z:Boolean) { y := (z:=true) and (z:=false) }
   ```

   ```
   compute(z:Boolean) { y := (z:=false) and (z:=true) }
   ```

3.3 Redundancy of Existing OCL Language Features

Some of the new language features in ImperativeOCL such as forEach and the imperative conditional are not really necessary (as long as ImperativeExpression is a subclass of OclExpression). Their effect 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
```

3.4 Further Problems

Apart from the problems illustrated above, we can find several other locations where allowing imperative expressions does not make sense. For example, ImperativeOCL would allow us to modify the system state in an invariant or a post-condition. The evaluation of the invariant could evaluate to true and invalidate the state at the same time.

Apart from the structural problems discussed above, we have the opinion that the subtype relation between ImperativeExpression and OclExpression violates the substitutability of subtypes for supertypes (e.g., [LW94]) very clearly. An imperative expression cannot be used everywhere a OCL expression is expected. On the contrary, there are only very few locations where imperative expressions can be safely used where an OCL expression is expected. Therefore, we propose a change to the QVT specification that leaves the semantics of conventional OCL unchanged and reuses OCL (as is) as a *part* of QVT's imperative language instead.

4 Suggested Change to the QVT specification

We think that ImperativeOCL expressions have not been intended to be used at all locations where OCL expressions occur. Therefore, imperative languages such as the one defined in QVT (called ImperativeOCL at the moment) should use OCL by composition rather than by inheritance, as depicted in Fig. 3.

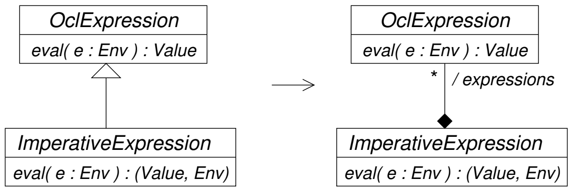

Fig. 3. Suggested change to the abstract syntax of QVT

Several concrete changes in the abstract syntax of ImperativeOCL follow from this modification. Most important, two versions of assignment expressions will be required: one whose right-hand side is of type OclExpression (as current) and one whose right-hand side is an ImperativeExpression (to capture the result of a compute or instantiation expression). Figure 4 shows the modifications to the abstract syntax class diagram.

The body of imperative loops will not longer be inherited from the OCL meta-class LoopExp. Iterators and the (imperative) body expression are modeled explicitly now (Fig. 5).

Similar changes have to be made for conditional execution (meta-classes AltExp and SwitchExp), while loops (meta-class WhileExp), and general block expression (meta-class BlockExp).

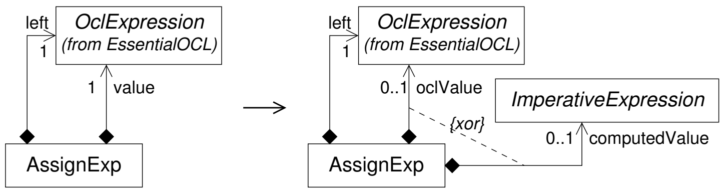

Fig. 4. Suggested change to meta-class AssignExp

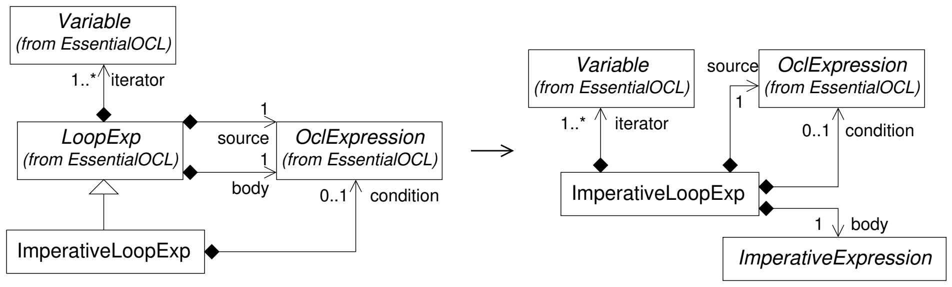

Fig. 5. Suggested change to meta-class ImperativeLoopExp

The sketched modification makes a clear distinction between imperative and logical language elements (i.e., conventional OCL). This solves all of the aforementioned problems: For the OCL part (then unchanged from [OMG06]), no under-definedness is introduced and the expected equivalence rules hold again. Also, it is made clear that no expressions with side-effects can occur at unexpected locations such as invariants and post-conditions. Imperative loops and conditional execution are clearly separated from the logical versions.

However, constellations such as the one provided in the introduction of this paper are not longer possible if the abstract syntax of ImperativeOCL is changed this way. While imperative expressions can still contain OCL expressions, OCL expressions can no longer contain imperative parts to calculate sub-results:

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

will be no valid expression. The example would have to be reworked either as pure OCL for the right-hand side of the assignment

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

or into a fully imperative version (except the arithmetic and relation expressions that are OCL):

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

Of course, this is a very simplified example. Imperative expressions in real QVT applications may be more complicated to rewrite. Especially, if imperative operations are invoked as part of an expression. For example the following imperative expression, using an operation with side-effects (calcAgeImperatively)

```
if (calcAgeImperatively(p1) > calcAgeImperatively(p2)) {...}
```

would have to be rewritten as

```
var ageOfP1 : Integer = calcAgeImperatively(p1);
var ageOfP2 : Integer = calcAgeImperatively(p2);
if (ageOfP1 > ageOfP2) { ... }
```

because the arguments of the relational OCL expression cannot be imperative expressions.

5 Conclusion

In this paper, we have pointed out a couple of semantical problems that all arise from the way OCL is embedded into QVT's ImperativeOCL. The design which subclasses OclExpression in the abstract syntax does not allow to replace subtype instances for supertype instances. It also requires an extended semantics of all conventional OCL expressions which is not defined at the moment.

We outlined a change to ImperativeOCL that resolves these problems by reusing OCL (as it is) in a non-intrusive way, making OCL a part of the imperative language. While this change requires certain (intermixed) expressions to be rewritten, it essentially reduces the complexity of the imperative language, removes undefinedness, and leaves OCL conformant to its original definition.

References

[ABJK06] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev. Atl - eclipse support for model transformation. In *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France*, 2006.

[AP05] Dave Akehurst and Octavian Patrascoiu. The Kent Modeling Framework (KMF). `http://www.cs.kent.ac.uk/projects/ocl`, University of Kent, 2005.

[COT08] Dan Chiorean and OCLE-Team. Object Constraint Language Environment 2.0. `http://lci.cs.ubbcluj.ro/ocle/`, 2008.

[DOT08] Dresden-OCL-Team. Dresden OCL Toolkit. `http://dresden-ocl.sourceforge. net/`, 2008.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[Kla05] Klasse Objecten. The Klasse Objecten OCL Checker Octopus. `www.klasse.nl/english/research/octopus-intro.html`, Klasse Objecten, 2005.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[MOT08] MDT-OCL-Team. MDT OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`, 2008.

[OMG06] Object Modeling Group. *Object Constraint Language Specification, version 2.0*, June 2006. OMG document formal/2006-05-01, `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[OMG08] Object Modeling Group. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*, 2008. OMG document formal/08-04-03, `http://www.omg.org/spec/MOF/2.0/PDF/`.

[Ric02] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[RT08] RoclET-Team. Welcome to RoclET. `http://www.roclet.org/`, 2008.