Modular Embedding of the Object Constraint Language into a Programming Language

Fabian Büttner and Martin Gogolla

University of Bremen, Computer Science Department, Database Systems Group
{green,gogolla}@tzi.de

Abstract. The Object Constraint Language (OCL) is a well-accepted ingredient in model-driven engineering and accompanying modeling languages like UML (Unified Modeling Language) or EMF (Eclipse Modeling Framework) which support object-oriented software development. Among various possibilities, OCL offers the formulation of state invariants and operation contracts in form of pre- and postconditions. With OCL, side effect free query operations can be implemented. However, for operations changing the system state an implementation cannot be given within OCL. In order to fill this gap, this paper proposes the language SOIL (Simple OCL-like Imperative Language). The expression sub-language of SOIL is identical to OCL. SOIL adds well-known, traditional imperative constructs. Thus by employing OCL and SOIL, it is possible to describe any operation in a declarative way and in an operational way on the modeling level without going into the details of a conventional programming language. In contrast to other similar approaches, the embedding of OCL into SOIL is done in a new, careful way so that elementary properties in OCL are preserved (for example, commutativity of logical conjunction). The paper discusses the central principles behind this conservative embedding of OCL into SOIL. SOIL has a sound formal semantics and is implemented in the UML and OCL tool USE (UML-based Specification Environment).

1 Introduction

Modeling languages like UML (Unified Modeling Language) or EMF (Eclipse Modeling Framework) play a central role in object-oriented software development and rely on a model-centric approach for development in contrast to traditional code-centric approaches. One main idea when using models is to find and to formulate central structural and behavioral properties of the system under development in an abstract, implementation independent way. Visual modeling notations are typically enriched by the textual Object Constraint Language (OCL) [WK03,CW02] which combines elements of first order predicate logic with object navigation. OCL allows the developer to formulate properties of a model that cannot be expressed in the visual notation. Typical applications of OCL are the formulation of class invariants (to express structural properties) and pre- and postconditions for operations as well as guards for state charts (to express behavioral properties).

While there are several visual possibilities like state charts or activity diagrams for modeling behavior, there is no way which allows the developer to express imperative algorithms in textual form. However, there are two important areas that require such a complementary textual notation for models, because they involve a considerable number of imperative algorithms: executable models and model transformations. Both areas typically combine a visual notation like state charts and graph transformations with imperative formulations of algorithms.

- For example, the Executable UML approach [MSUW04,Mel02] describes UML models that can be actually executed. This is achieved by providing Moore state charts for all operations of the model. Furthermore, a textual action language is used to describe the effects of the states in the state machines.
- A second application of textual imperative languages within modeling is found in the Model-Driven Architecture (MDA) [OMG03]. The OMG Query, Views, Transformation (QVT) specification [OMG08] describes several layers to transform abstract models into more specific models. At its core, it offers a textual imperative language, ImperativeOCL, which is based on OCL.
- A third area, where textual imperative descriptions within models are needed, are operations which change the state of the system under development. It is possible to formulate so-called query operations with textual OCL. Such operations do not modify the system state but retrieve other objects and data from source objects. However, it is not possible to formulate the implementation of state changing operations in OCL whereas one can determine the behavior of such operations by stating pre- and postconditions.

Thus, one the one hand side, there is OCL, which has already proven to be a valuable expression language with broad support of tools. On the other hand, there are several areas that require an imperative language for and within models. Since imperative statements like assignments or conditionals require expressions, the idea to use OCL as an expression language within an imperative programming language naturally comes up.

Indeed, there is a number of imperative languages that reuse OCL as an expression language. However, if one looks on the reuse of OCL in these approaches in depth, there are different understandings of reuse. ImperativeOCL is an example for a kind of weak OCL reuse that prohibits the direct reuse of OCL tools and it is also an example for a language that introduces semantic problems for OCL expressions [BK09]. As an alternative, we developed the language SOIL (Simple OCL-based Imperative Language) [Büt11]. SOIL has a sound formal semantics and is type safe. SOIL is implemented in the UML-based Specification Environment (USE) [GBR07].

In the present paper, we will use SOIL as a showcase to illustrate the major criteria for reusing OCL in an imperative programming language. The rest of the paper is structured as follows. We first motivate why OCL could be reused in an imperative programming language in Sect. 2. In Sect. 3, we give an example of how SOIL is applied to specify the semantics for operations in models. In Sect. 4 we discuss general criteria for the reuse of OCL in an imperative programming language. When necessary, we refer to SOIL

to illustrate a modular kind of reuse. Section 5 shortly highlights the consequences of a modular embedding in terms of language expressiveness. We conclude our paper in Sect. 6.

2 Motivation for Reusing OCL

In the context of model-driven engineering and model-transformation, there are several reasons to reuse OCL. Formal approaches such as Executable UML and MOF QVT require precise operational descriptions that cannot always be expressed reasonably using only visual notation. Textual imperative languages are required to fill this gap. The official OMG language ImperativeOCL extends OCL by so-called imperative expressions to suit this need. Other approaches combining OCL or an OCL-like language with imperative programming are EOL [KPP06] and OCL4X [JZM08].

There are several reasons to build these languages on top of OCL. First of all, we can assume OCL to be already known in context where these languages are used. Developers familiar with modeling languages typically know OCL already. Thus, learning the respective imperative language becomes easier when the expression language is already known. This is true in particular as these languages are typically rather lightweight. They do not aim to compete against general purpose languages such as Java or C#.

Another reason for reusing OCL is the possibility to reuse existing OCL tools: The implementation of a programming language based on OCL can be simplified if one can avoid to deal with expressions again. The infrastructure for UML and EMF models and OCL expressions is already available in several tools. The long list of publicly available OCL tools includes the Dresden OCL toolkit [HDF02], the OCL Environment (OCLE) [CPC+04], the ATL tool [JABK08], the Eclipse Model Development Tools (MDT) Eclipse MDT OCL [MDT], KMF [AP08], the Octopus tool [Kla05], RoclET [RT], and the UML-based Specification Environment (USE) [GBR07].

Furthermore, the scientific community has developed a number of formal approaches that deal with OCL expressions and OCL-annotated models. These approaches include expression transformation (e. g., in [MB08,CT07,Büt05]), expression analysis (e. g., in [CJMB08]), reasoning (e. g., in [BW08]), and model checking (e. g., in [DKR00,KK08]). These results can be employed further, if OCL is used as an expression language within an imperative programming language.

For these reasons, we think that there are strong arguments for reusing OCL in imperative programming. However, as we will point out in Sect. 4, we have to be careful in the definition of an OCL-based imperative language. OCL has to be embedded in a modular way when one wants to take advantage of the mentioned profits.

3 SOIL by Example

In this section we give a concrete example for using an imperative programming language for UML models: The language SOIL as part of the UML-based Specification

Environment (USE). USE supports the modeler in two ways: (1) prototypical model states for OCL-annotated UML models can be validated against structural constraints (including OCL invariants), and (2) prototypical model executions can be validated against dynamic constraints (i. e., OCL pre- and postconditions). Previously, there was no universal means to specify imperative programs for OCL-annotated models employing general loops, operation calls and recursion. This gap was filled by SOIL. Using SOIL, imperative definitions can be given for the operations of a model, and the imperative definitions can be validated against the structural and dynamic constraints of the model. The language has a formally defined semantics and the type system is proved to be sound [Büt11].

The extended USE tool now enables stepwise refinement from a declarative model (pre- and postconditions) towards an operational model (operation implementation) in an integrated model-based environment. The following short example shows how SOIL is used to perform this refinement step. Consider the class diagram in Fig. 1. In this project world, companies employ workers and carry out projects. Workers bring certain qualifications (e.g., programming) and projects require certain qualifications. In order for a project to become active, it must have members for all required qualifications. In this class diagram, we have only one non-query operation, schedule(), to assign workers to projects. A good implementation of schedule() will ensure a good use of the company's human resources (ideally, carry out as much projects as possible).

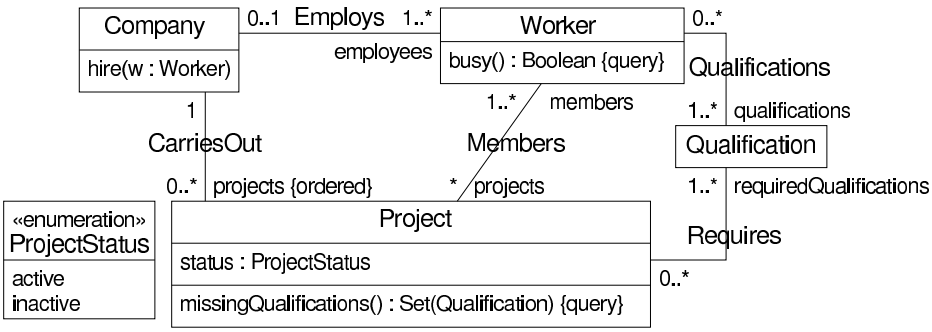

Fig. 1. Project World

Some properties of this operation are further specified in a declarative way by OCL postconditions as shown in Listing 1.1: After scheduling projects, it has to be ensured that no active project misses any qualifications and no employee is working in two active projects at the same time. The listing also shows the definition of the two query operations missingQualifications() and busy(). These side effect free operations are defined straightforward by OCL expressions.

```
context Project def: missingQualifications() : Set(Qualification) =
  self.requiredQualifications - self.members.qualifications->asSet

context Worker def: busy() : Boolean =
  self.projects->exists(p | p.status = #active)

context Company::schedule()
  post activeProjectsHaveRequiredQualifications:
    self.projects->forAll(p | p.status = #active implies
      p.missingQualifications()->isEmpty)
  post employeesNotOverloaded:
    self.employees->forAll( w | w.projects->select(p |
      p.status = #active)->size <= 1)
```

Listing 1.1. Declarative specification of Company::schedule

Obviously, several implementations of schedule() will full the above pre- and postconditions. The USE tool allows us define schedule() using SOIL statements. Giving an initial state, all SOIL defined operations can be invoked. Recursive invocation is supported, as well. During the animation of the model, all structural and dynamic constraints are checked. In our example, the execution of the schedule() operation is validated against the above postconditions. We can compare this functionality to programming languages that support design-by-contract (such as Eiffel [Mey92]). However, in our case we are still in the context of the UML object model. In particular, OCL expressions can be used within our imperative definition.

Listing 1.2 shows a very simple imperative version of schedule(). We can see that the SOIL provides typical flow control constructs (for-loop, if-statement). Within these statements, OCL expressions are used to describe the parameters (e.g., the range for the iteration and the condition for the if-statement. Statements to manipulate the system state are available (in the above example: link insertion and attribute assignment). The semantics of these statements is straightforward.

```
context Company def: schedule() =
  for w in self.employees do
    for p in self.projects do
      if p.missingQualifications()
           ->intersection(w.qualifications)->notEmpty then
        insert (p, w) into Members;
        if p.missingQualifications()->isEmpty and not w.busy() then
          p.status := #active
        end
      end
    end
  end
```

Listing 1.2. Operational specification of Company::schedule

While this very simple implementation of schedule() conforms to the postconditions, it is not an optimal implementation, since it will not result in a maximum number of projects being active. We can construct a more sophisticated version that schedules and

reschedules project members to achieve an optimal number of active projects. However, such an implementation is much more complex and, therefore, error prone. The integrated descriptive and operation specification in USE with OCL and SOIL allows a smooth and step-wise refinement process from the descriptive model to an actual imperative model, guiding the developer by validating the models through animation against the constraints.

4 Embedding of OCL into SOIL

In the previous section, we gave an example for an imperative programming language for UML models. As several other related languages such as ImperativeOCL, it reuses OCL for expressions. However, not all of these languages realize the benefits we gave as reasons for reusing OCL in Sect. 2.

In this section, we now discuss several concepts of imperative programming languages from the perspective of reusing OCL. We start with a short recapitulation of the formalization of OCL expressions, then we inspect the amalgamation of statements, local variables, operation invocation, and state manipulation. As we will see, the major pitfalls in a successful modular reuse of OCL lurk in the amalgamation of statements and expressions and in an indeterminate treatment of operations with side effects and query operations.

For each of the mentioned concept we provide the corresponding piece of SOIL to illustrate a safe and modular reuse. A complete guide to SOIL, including a formal definition of the language as well as proofs for the type soundness can be found in [Büt11].

4.1 Formal Representation of OCL Expressions

We shortly sketch the formal definitions for UML static structure models and OCL expressions. These definitions have been originally provided in [Ric02] and are now enclosed in the OCL specification [OMG06]. For the objective of this paper, a complete depiction of the formalization is not necessary as we only need the general concepts in the following.

The object model

$$\mathcal{M} = (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{ASSOC}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

is the formal representation of the major concepts UML provides for static structure modeling (say class diagrams). It contains all classes along with their attributes, operation signatures, associations, and generalization relationships. The set μ denotes the set of all instances of $\mathcal{M}$. Thus, a system state $\sigma \in \mu$ describes a set of objects, links, and attribute values.

For the formalization of OCL expressions, we require a data signature over $\mathcal{M}$ which is a structure

$$\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$$

where $T_{\mathcal{M}}$ is the set of all types over $\mathcal{M}$. This includes primitive types, user types (in particular, classes), and all collection types can be constructed by the OCL collection type constructors. The relation $\leq$ is the type hierarchy over $T_{\mathcal{M}}$. The set $\Omega_{\mathcal{M}}$ contains the set of all query operations (operations without side effects), and thus corresponds to a subset of OP_c.

The semantics of $\Sigma_{\mathcal{M}}$ is as follows. $I(T_{\mathcal{M}})$ assigns each type $t \in T_{\mathcal{M}}$ an interpretation $I(t)$ (the domain of t). $I(\leq)$ implies for all types $t', t \in T_{\mathcal{M}}$ that $I(t') \subseteq I(t)$ if $t' \leq t$. $I(\Omega_{\mathcal{M}})$ assigns each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ a total function $I(\omega) : \sigma \times I(t_1) \times \cdots \times I(t_n) \to I(t)$.

Given the data signature $\Sigma_{\mathcal{M}}$, we can formalize the set Expr of all OCL expressions that exists over $\Sigma_{\mathcal{M}}$. For each expression $e \in$ Expr, the function free : Expr $\to$ Var determines the free variables of e (Var being the set of all typed variables).

The interpretation of an expression $e \in$ Expr is given by a function $I[\![\, e \,]\!]$ which assigns a value to each pair $\tau = (\sigma, \beta)$ of a system state σ of $\mathcal{M}$ and a variable assignment β : Var $\to I(t)$.

4.2 Statements

Imperative programming languages typically refer to their smallest standalone elements as statements. The effect of such a statement is determined by the effect is has on the process environment (the state). For imperative languages that work on object models, the state at least contains the available objects, links, and attribute values, as well as a representation of the variable assignments.

If we want to describe the semantics of an imperative language in a similar fashion as for OCL, then we have to describe the interpretation for each statement s of that language by an interpretation function. A minimalistic interpretation function for statements is a function $I[\![\, s \,]\!]$ which assigns each pair (σ, β) of a system state σ and a variable assignment β a new pair (σ', β'). If we furthermore want a statement to have a value in a functional sense, we require an interpretation that assigns each pair (σ, β) a triple (σ', β', y) where y is the functional value of statement s.

Statements having a functional value may also occur where an expression is expected. Several statements in common programming languages have this kind of semantics, for example the assignment statement $b = a$ in Java which (1) leads to a new environment (with b having a new value) and (2) has a functional value (the value of a). It can be used as an expression as well, therefore a statement like $c = (b = a)$ is valid in several programming languages.

However, for a modular reuse of OCL, it is important to keep statements and OCL expressions clearly separated. We will use the language ImperativeOCL to illustrate the problems that result from an amalgamation of statements and OCL expressions.

ImperativeOCL defines several new kinds of OCL expressions. These new expressions are called imperative expressions and have a combined functional resp. imperative se-

mantics as explained above. In the ImperativeOCL metamodel, the imperative expressions are introduced as subclasses of *OclExpression* (and therefore, imperative expressions extend the set of OCL expressions).

In particular, the *compute* expression can be used to capture the result of a sequence of imperative statements as a functional value. In ImperativeOCL, the following expression has the value 6 ($1 + 2 + 3$):

```
1 + compute(b : Integer) { a := 1; b := a + 1 } + 3
```

The compute expression declares a local variable and contains a sequence of imperative expressions. The value 2 of the above compute expression is determined by the final value of b after executing the statements of the body. If we assume the second variable a to be declared somewhere before, the compute expression also has an effect that is visible outside the compute expression, as a (possibly) new value (1) will be assigned to a after the evaluation of the compute expression.

Now we use a more complex example. Assume *true* has been assigned to the variables a and b before, and notice that the imperative assignment expression $x := y$ of ImperativeOCL has the same value semantics as discussed above:

```
compute(c:Boolean) { if ((a:=false) and (b:=false)) { ... }; c := a  }
```

The value of this compute expression is false (it returns the value of c at the end of the block). The interpretation, however, becomes less obvious if we change the last assignment:

```
compute(c:Boolean) { if ((a:=false) and (b:=false)) { ... }; c := b }
```

The interpretation of this compute expression 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 $I[\![e_1 \text{ and } e_2]\!](\sigma, \beta)$:

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

$$I[\![e_1 \text{ and } e_2]\!](\sigma, \beta) = \begin{cases} I[\![e_2]\!](\sigma', \beta') & \text{if } y = \text{true} \\ (\sigma', \beta', y) & \text{otherwise} \end{cases}$$

   where $(\sigma', \beta', y) = I[\![e_1]\!](\sigma, \beta)$. Under this semantics (also called short-circuit evaluation) the right-hand side of the *and* operator is not evaluated unless the left-hand side evaluates to *true*. Therefore, b stays *true*.
2. Strict evaluation semantics (returns false for the above example):

$$I[\![e_1 \text{ and } e_2]\!](\sigma, \beta) = \begin{cases} (\sigma'', \beta'', \text{true}) & \text{if } y_1 = \text{true} \wedge y_2 = \text{true} \\ (\sigma'', \beta'', \text{false}) & \text{otherwise} \end{cases}$$

   where $(\sigma', \beta', y_1) = I[\![e_1]\!](\sigma, \beta)$ and $(\sigma'', \beta'', y_2) = I[\![e_2]\!](\sigma', \beta')$. Under this semantics, both sides of the *and* operator are always evaluated. Therefore, *false* is assigned to b.

There is no rule on short-circuit evaluation in OCL. OCL, which can be regarded as a kind of first order predicate logic, does not need such a rule. An optimizing OCL compiler might even decide to short-circuit evaluate the second operand first if this seems reasonable.

However, in order to have a clear semantics, ImperativeOCL implicitly requires a decision on this question. Similar issues regard the commutativity of operators etc. Of course these decisions can be made for ImperativeOCL, but they may be inappropriate for other applications of OCL. And, existing OCL tools may have differing implementations and may be therefore unusable to implement ImperativeOCL.

A more general argument against the amalgamation of statements and expressions is that OCL expressions are no longer side effect free by introducing *ImperativeExpression* as a subclass of *OclExpression*. In our understanding, this breaks a fundamental property of the *OclExpression* class. Therefore the ImperativeOCL metamodel breaks the subtype substitution principle. The direct result is that formal approaches such as expression transformations, expression analysis, reasoning, and model checking cannot longer be applied to OCL expressions in the context of the ImperativeOCL extension.

Therefore, we require a strict distinction of statements and OCL expressions for a modular reuse of OCL. Fig. 2 depicts this requirement on the level of the language metamodels. Notice that, from the perspective of modular reuse, an imperative programming language might add further kinds of expressions which are not OCL. However, these expressions must not occur as OCL expressions.

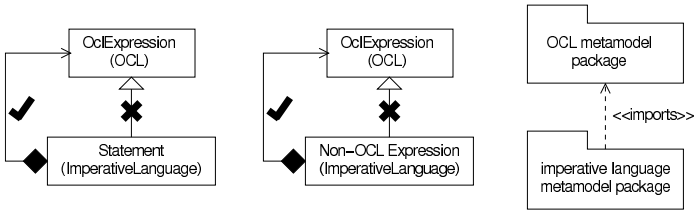

Fig. 2. Modular Embedding

A similar argumentation for composition and modularity of domain specific languages can be found in [KRV08] and [Hud98]. It is also aligned with [MSS04] in the sense that side effected non-modular extensions of OCL should be avoided.

SOIL Example For the reasons given, statements are clearly separated from OCL expressions in SOIL. To illustrate the formalization of statements in SOIL, we show how the syntax and semantics of the imperative *if-then-else* are defined.

The syntax is defined as follows[1]: If $e \in \text{Expr}_{\text{Boolean}}$ and $s_1, s_2 \in$ Stmt then

$$\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end} \in \text{Stmt}.$$

We can see that this kind of statement *contains* an expression. But despite syntactic similarities to the functional if-then-else of OCL, the imperative if-then-else is a completely different entity: The definitions of Expr and $I[\![e]\!]$ are not changed or extended by SOIL. The imperative language defines a new set of statements Stmt, which is disjoint with Expr.

The meaning of each statement $s \in$ Stmt is given by an interpretation function $I[\![s]\!]$ which assigns each pair (σ, ζ) of a system state and a variable assignment a new pair (σ', ζ'). Notice that, for technical reasons, we distinguish the imperative variable assignments ζ (which actually are a stack structure) and the variable assignments β used to evaluate OCL expressions).

The semantics of the *if* statement is defined as follows.

$$I[\![\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end}]\!](\sigma, \zeta) := \begin{cases} I[\![s_1]\!](\sigma, \zeta) & \text{if } I[\![e]\!](\sigma, \text{binding}(\zeta)) = \text{true} \\ I[\![s_2]\!](\sigma, \zeta) & \text{otherwise} \end{cases}$$

Corresponding to the syntactic containment of OCL expressions as part of statements, the interpretation function for OCL expressions occurs within the above definition of the interpretation function for the *if* statement. The condition expression e is evaluated in the same context (state, variables) as the statement. To pass the variable assignments from $I[\![s]\!]$ to $I[\![e]\!]$ we require a transformation binding to map between the different notions of variable assignments in SOIL and OCL (see next subsection).

All kinds of statements in SOIL are defined in this manner.

4.3 Local Variables

Variable assignment is a core concept available in all imperative programming language. When statements contain OCL expressions, the assignments of previous statements will be visible for the evaluation of OCL expressions in subsequent statements. Consider the imperative program:

```
1  a := 1; b := a + 1.
```

The OCL expression in the second statement has one free variable a. A value for a will be available after the execution of the first statement (as a furthermore has the right type, the above concatenation of statements is even type sound). This relationship is depicted in Fig. 3. In SOIL the mapping from the imperative variable environment (which is a stack) to the variable assignment (which is a flat mapping) required for the evaluation of OCL expressions is realized by the *binding* operation which already occurred above.

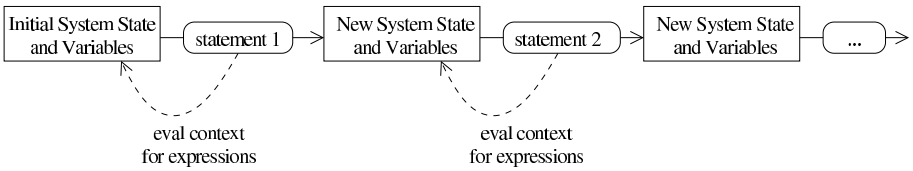

Fig. 3. Evaluation Chain for Statements and Expressions

Technically, *binding* makes the assignments in the top-most stack frame in ζ available as a flat mapping from typed variables to values.

There are no particular obstacles regarding local variables w.r.t. modularity of OCL. However, if we want static type checking, it is important to ensure that correctly typed values are available for all free variables in the OCL expressions that are part of our statements (we provide such a type system in SOIL).

4.4 Operations with Side Effects

The application of operations with side effects within OCL expressions constitutes a similar problem as the amalgamation of statements and OCL expressions. While the interpretation of a query operation is a value (see Sect. 4.1), the interpretation of an operation with side effects yields a new state (and possibly a value). For a modular reuse of OCL we cannot allow the second one to occur in OCL expressions.

In order not to stretch short-circuit evaluation or commutativity of relational operations for the explanation, again, we take a look on the *let* expression in OCL. This expression substitutes an expression for a variable. As for predicate logic, the following important equivalence rule holds for OCL:

$$I[\![\,\text{let } v : T = e_1 \text{ in } e_2\,]\!](\sigma,\beta) = I[\![\,e_2\{v/e_1\}\,]\!](\sigma,\beta).$$

However, this rule is broken if we allow operations with side effects within OCL expressions. Assume a class Person with attributes firstName and lastName. Consider an operation newPerson():

```
1  def: newPerson(firstName : String, lastName : String):Person =
2    w := new Worker;
3    w.firstName := w.firstName;
4    w.lastName := w.lastName;
5    return w
```

Obviously, the interpretation of

[1] Notice that we omit all typing rules in this paper as they are not relevant for the discussion of modularity. The full definition of SOIL provides typing rules to ensure that all statements are statically type safe.

```
let w : Worker = newWorker('Bob', 'Builder') in
  w.lastName.concat(', ').concat(w.firstName)
```

is different from the interpretation of

```
newWorker('Bob', 'Builder').lastName.concat(', ').concat(
  newWorker('Bob', 'Builder').firstName)
```

which will create two Worker objects.

As mentioned above, these problems can be constructed in several ways if we allow operations with side effects in OCL expressions. Therefore, we require a distinction between query operations and operations with side effects. Within OCL expression, only query operations must be used. Otherwise, we run into the same problems mentioned in Sect. 4.2. Consequently, an imperative language must include a dedicated means to invoke operations with side effects.

SOIL Example The language provides specific statements to invoke operations with side effects. Here, we show the form which invokes an operation that has a return value (another statement is available to invoke operations without return value). Its syntax is as follows: If $e_1 \in \text{Expr}_{t_1}, \ldots, e_n \in \text{Expr}_{t_n}$, $v \in$ Varname, and $\overline{\omega} : (v_1 : t_1, \ldots, v_n : t_n \to t) \in \overline{\Omega}_{\mathcal{M}}$ then

$$v := \overline{\omega}(e_1, \ldots, e_n) \in \text{Stmt}.$$

The most important point here is that the operation to be invoked has to be in the set of the operations with side effects $\overline{\Omega}_{\mathcal{M}}$, whereas query operations (which can be occur in OCL expressions) are in $\Omega_{\mathcal{M}}$ (c.f. Sect. 4.1).

Given Z being the set of all variable assignments ζ, the semantics of each $\overline{\omega} : t_1 \times \cdots \times t_n$ in $\overline{\Omega}_{\mathcal{M}}$ is a total function

$$I(\overline{\omega}) : \mu \times Z \times I(t_1) \times \cdots \times I(t_n) \to \mu \times Z$$

that assigns to the current system state, variable assignments, and parameters a new system state and a new variable assignment.

The semantics of s is then given as follows. Let $x_1, \ldots, x_n = I[\![e_1]\!](\sigma, \text{binding}(\zeta)), \ldots, I[\![e_n]\!](\sigma, \text{binding}(\zeta))$, then

$$I[\![v := \overline{\omega}(e_1, \ldots, e_n)]\!](\sigma, \zeta) := (\sigma', \zeta'\{v/z\})$$

where $(\sigma', \zeta', z) = I(\overline{\omega})(\sigma, \zeta, x_1, \ldots, x_n)$.

4.5 State Manipulation Statements

Imperative languages that operates on UML models typically at least provide the following capabilities: object creation and destruction (unless a garbage collection approach is applied), link manipulation, and attribute assignment. Most of these statements have parameters (e.g., to determine the elements of a link) which can be given

by OCL expressions. If the modularity aspects highlighted so far are obeyed, there are no further obstacles w.r.t to a modular reuse of OCL. As for local variables, state manipulations in a previous statement have to be visible in OCL expressions as part of a subsequent statement.

5 Consequences of a Modular Embedding

In the previous sections we discussed several pitfalls for a modular embedding of OCL. If we avoid these pitfalls, we can achieve benefits stated in Sect. 2. Apart from the syntactical differences, languages that reuse OCL in a modular way (such as SOIL) can be able to express programs in a similar way as languages that do not reuse OCL that way, like ImperativeOCL. Comparing SOIL and ImperativeOCL, both languages provide the full power of OCL for expressions.

There are, however, kinds of statements that cannot be translated one-to-one from ImperativeOCL to SOIL or to any language that obeys the rules given in the previous section. Specifically, these statements are statements that contain expressions that contain statements. Constructions such as

```
1  mySeq := Sequence{1,2,3}->collect( x |
2     compute(y:Integer) {
3       y := 0; Sequence{1..x}->forEach(z){ y := y + z }
4     })
```

are not possible in SOIL and have to be decomposed into several steps in SOIL:

```
1  mySeq := Sequence(Integer){};
2  for x in Sequence{1,2,3} do
3    y := 0; for z in Sequence{1..x} do y := y + z end;
4    mySeq := mySeq->append(y)
5  end
```

Such amalgamation of expressions and statements have to be resolved in several steps in a modular embedding of OCL. Notice that this includes invocations of non-query (i.e., side effected) operations from OCL expressions: Assuming f and g to be operations with side effects that furthermore yield integer values, the following ImperativeOCL expression

```
1  result := f() + g() + 1
```

has to be rewritten in SOIL to

```
1  fVal := f();
2  gVal := g();
3  result := fVal + gVal + 1.
```

Of course a imperative language might allow the upper syntax as a shortcut for the lower syntax, but it is important to see that this effectively introduces a new set of non-OCL expressions as part of that imperative language (as depicted in the middle part of Fig. 2 on the metamodel level). While the syntax might look the same as OCL, existing OCL

compilers (or interpreters) cannot be used to implement it, nor can we reuse other formal approaches for OCL expressions, for the reasons given in Sect. 3. Of course, one might believe this redundant approach to be viable for simple arithmetic expressions as above. However, we cannot see where to draw the line here: If we want to allow operations with side effects anywhere in a right-hand side expression of an assignment statement (for example), we have to re-implement the whole OCL syntax for this custom expression language (i. e., we don't anymore reuse OCL in the sense of Sect. 2). If we only allow certain (say, simple) expressions such as arithmetic expressions, it will probably appear inconsistent and confusing to the modeler, as operations with side-effects are allowed in some expressions only.

For these reasons, we believe that such a redundant approach should be avoided completely. The resulting general restrictions are the price we have to pay for a language that reuses OCL in a modular and comprehensive way. In the scope of programming (with) models, we think that the benefits by far outweigh this price. This holds in particular if we consider that we already have the full power of OCL expressions as part of the imperative language and therefore a lot of programming can be done in a functional manner.

6 Conclusion

In this paper we presented our understanding of a modular reuse of OCL in an imperative language. If languages embed OCL this way, the reuse of existing tools and libraries, of knowledge that developer already gained for OCL, and of formal methods for OCL expressions, is possible. Several OCL-based, or OCL-inspired languages fail to fulfill this requirements, in particular ImperativeOCL.

Based on this observation we developed the language SOIL. SOIL is a simple and unspectacular but complete imperative language that can be used to operationally specify UML models (i. e., to program (with) UML models). We used SOIL to illustrate the major drawbacks in the design of OCL-based imperative languages.

The intrinsic drawbacks of SOIL (and any other language that is based on OCL in a modular way) w. r. t. to monolithic languages such as ImperativeOCL regard amalgamation of expressions and statements. These constructs have to be decomposed in SOIL, which, in general, leads to larger programs. We believe, however, that for most of the (rather restricted) scenarios of programming with models, the benefits of reusing the well-known and established language OCL outweigh these extra efforts.

A number of topics will be addressed in future work. SOIL has already been employed in smaller case studies, but larger case studies must give feedback on the usability and efficiency of the language. Further imperative constructs like more convenient loops and error handling should be addressed. SOIL is compliant with the UML Actions metamodel. Therefore, it could be used, in the Executable UML approach, in conjunction with state machines in order to create fully executable descriptions of a system.

References

[AP08] Dave Akehurst and Octavian Patrascoiu. KMF (Kent Modeling Framework) OCL Library. website, http://www.cs.kent.ac.uk/projects/ocl/tools.html, last visited 10.02.2011, 2008.

[BK09] Fabian Büttner and Mirco Kuhlmann. Shortcomings of the Embedding of OCL into QVT ImperativeOCL. In Michel R.V. Chaudron, editor, *Workshops and Symposia at 11th Int. Conf. Model Driven Engineering Languages and Systems (MODELS'2008)*, pages 263–272. Springer, Berlin, LNCS 5421, 2009.

[Büt11] Fabian Büttner. *Reusing OCL in the Definition of Imperative Languages*. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, 2011.

[Büt05] Fabian Büttner. Transformation-Based Structure Model Evolution. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS'2005 Conference*, pages 339–340. Springer, Berlin, LNCS 3844, 2005.

[BW08] Achim D. Brucker and Burkhart Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer, 2008.

[CJMB08] Jesús Sánchez Cuadrado, Frédéric Jouault, Jesús García Molina, and Jean Bézivin. Deriving OCL Optimization Patterns from Benchmarks. *ECEASST*, 15, 2008.

[CPC+04] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML Models Consistency Using the OCL Environment. *Electronic Notes in Theorethical Computer Science*, 102:99–110, 2004.

[CT07] Jordi Cabot and Ernest Teniente. Transformation techniques for OCL constraints. *Science of Computer Programming*, 68(3):179–195, 2007.

[CW02] Tony Clark and Jos Warmer, editors. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[DKR00] D. S. Distefano, J. P. Katoen, and A. Rensink. Towards model checking OCL. In *ECOOP 2000: Defining Precise Semantics for UML, Sophia Antipolis, France*, June 2000.

[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.

[HDF02] Heinrich Hußmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, 44(1):51–69, 2002.

[Hud98] Paul Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

[JZM08] Ke Jiang, Lei Zhang, and Shigeru Miyake. Using OCL in Executable UML. *ECEASST*, 9, 2008.

[KK08] Matthias P. Krieger and Alexander Knapp. Executing Underspecified OCL Operation Contracts with a SAT Solver. *ECEASST*, 15, 2008.

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

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2006.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: Modular Development of Textual Domain Specific Languages. In Richard F. Paige and Bertrand Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer, 2008.

[MB08] Slavisa Markovic and Thomas Baar. Refactoring OCL annotated UML class diagrams. *Software and System Modeling*, 7(1):25–47, 2008.

[MDT] Eclipse model development tools (MDT) project page. Website, http://www.eclipse.org/modeling/mdt/, last visited 10.02.2011.

[Mel02] Stephen J. Mellor. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[MSS04] Jari Peltonen Mika Siikarla and Petri Selonen. Combining OCL and Programming Languages for UML Model Processing. In P.H. Schmitt, editor, *Proceedings of the Workshop, OCL 2.0 – Industry Standard or Scientific Playground*, volume 102, pages 175–194. Elsevier, 2004.

[MSUW04] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, Boston, 2004.

[OMG03] OMG. *MDA Guide Version 1.0.1*. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, June 2003.

[OMG06] OMG. *Object Constraint Language Specification, version 2.0 (Document formal/2006-05-01)*, June 2006.

[OMG08] OMG. *Meta Object Facility (MOF) 2.0 Query/Views/Transformation Specification (Document formal/08-04-03)*. Object Management Group, Inc., Framingham, Mass., Internet: http://www.omg.org, 2008.

[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.

[RT] RoclET-Team. Welcome to RoclET. Website, http://www.roclet.org/, last visited 10.02.2011.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, August 2003.