Reusing OCL in a Programming Language

Fabian Büttner[a,*], Martin Gogolla[b]

[a]AtlanMod, École des Mines de Nantes - INRIA, Nantes, France

[b]Database Systems Group, University of Bremen, Germany

Abstract

The Object Constraint Language (OCL) is a well-accepted ingredient in model-driven engineering and accompanying modeling languages such as UML (Unified Modeling Language) and EMF (Eclipse Modeling Framework) which support object-oriented software development. Among various possibilities, OCL offers the formulation of class invariants and operation contracts in form of pre- and postconditions, and side effect-free query operations. Much research has been done on OCL and various mature implementations are available for it. OCL is also used as the foundation for several modeling-specific programming and transformation languages. However, an intrusive way of embedding OCL into these language hampers us when we want to benefit from the existing achievements for OCL. In response to this shortcoming, we propose the language SOIL (Simple OCL-like Imperative Language), which we implemented in the UML and OCL modeling tool USE to amend its declarative model validation features. The expression sub-language of SOIL is identical to OCL. SOIL adds imperative constructs for programming in the domain of models. 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 careful, non-intrusive way so that purity of OCL is preserved.

Keywords: OCL, Programming Language, UML Embedding

*This research was partially funded by the Nouvelles Équipes Program of the Pays de la Loire Region (France).

1. Introduction

Modeling languages like UML (Unified Modeling Language) [37] or EMF (Eclipse Modeling Framework) [44] play a key role in model-centric development 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) [38, 46] which provides a first-order logic-like query language for objects. 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 derived attributes and guards for state charts (to express behavioral properties).

Several other modeling-specific languages have been defined on top of OCL, in particular in the field of model transformation and executable models. Well-known OCL-based languages include ImperativeOCL [36] (the official imperative modeling language of the OMG), the ATL transformation language [19, 20], the Epsilon Object Language (EOL) [22], OCL for Execution (OCL4X) [18], and the upcoming Action Language for Foundational UML (fALF) [35]. Essentially, those languages embed OCL as an expression or query language. There are several reasons to do this, as OCL is widely accepted in both the development and the research communities. However, the way OCL is embedded in these approaches bears certain conflicts with respect to the side-effect free semantics of OCL. Furthermore, it prohibits an 'off-the-shelf' reuse of existing OCL tools and formal reasoning approaches for these languages.

In this work, which is an extended version of [5], we present the language SOIL (Simple OCL-based Imperative Language). SOIL is an imperative language specific to the animation of model instances. It is motivated by two objectives: (1) by the need to have a simple, imperative model animation language for the UML-based Specification Environment (USE) [13, 45], an interpreter and validator for OCL and (a subset of) UML, and (2) to study a safe embedding of OCL into such a language, on the level of the syntax and the denotational semantics of OCL, and on the implementation level. While SOIL is a complete programming language and sufficient for our requirements in the model animation and model transformation contexts, it is not a a general purpose language. It has no I/O, no modules, no concurrency, no higher-order features, and no exception handling.

We describe the syntax and semantics of the language and show how OCL can be safely embedded in it, not changing or extending the syntax or semantics of expressions from the OCL specification. Consequently, we can illustrate how our compositional definitions of the syntax and semantics of SOIL naturally correspond to analogous module dependencies at the implementation level (i.e., in USE), meaning that we could reuse an existing parser, type-checker, and interpreter for OCL without changes when we added SOIL into USE. Compared to the expressiveness of, e.g., ImperativeOCL, our approach comes with (in our view) acceptable trade-offs, in the sense that some constructions require intermediate results in SOIL which could be calculated in one statement in ImperativeOCL.

We want to stress that we particularly require that the OCL language specification remains unchanged. OCL could be advanced in many ways, e.g., by adding higher-order functions, that would provide safe (and more elegant) ways of embedding imperative languages on top of OCL (in particularly, using Monads). Our contribution makes no contribution in this direction, and our approach does not claim to provide new results in the field of programming language semantics. Since OCL is however a part of several industrial standards, and implemented in various tools, we see the need for a contribution that highlights the embedding problems and provides a simple example how they can be avoided (with OCL in its current shape).

Organization. In Sect. 2 we motivate a safe embedding of OCL and identify the major problems in the current approaches embedding OCL. In Sect. 3 we give an informal introduction to our language SOIL and the tool USE. Section 4 provides the necessary background on OCL that we need to describe SOIL precisely in Sect. 5. In Sect. 6 we discuss the results and illustrate how SOIL has been implemented in USE. In Sect. 7 we put our contribution in the context of related work, and we conclude in Sect. 8.

2. Reusing OCL

In the context of model-driven engineering, we find several programming and transformation languages that operate on models, and that require a corresponding model query language. The Object Constraint Language (OCL) is one de-facto standard for such a query language.

Often, model-based languages also require a certain amount of imperative features in order to express all programming aspects. In addition to model transformation languages, this also regards, for example, the Executable UML approach [30, 29]. Textual imperative languages are employed to fill this gap, and there are several good reasons to build these languages on top of OCL. In OMG QVT [36] the language ImperativeOCL extends OCL by so-called *imperative expressions* to suit this need. The ATL transformation language [19, 20] provides a set of statements that can be used to specify aspects of transformation rules in an imperative way. The upcoming Action Language for Foundational UML (fALF) [35], OCL for Execution (OCL4X) [18], and the Epsilon Object Language (EOL) [22] amend OCL with kinds of statements or action expressions in similar vein.

We have already stated a first reason to reuse OCL in these languages: It is well accepted by both the development and research communities. We can assume OCL to be already known in the context where these languages are used. This supports both the effort to develop the resp. OCL-based language and the effort to learn it. This is true in particular as these languages are typically rather lightweight and do not aim to compete against general purpose languages such as Java or C#. Thus, expressions typically make up a significant part of such lightweight languages.

A second reason for reusing OCL in these languages is the possibility to reuse existing implementations of OCL, resp., libraries. The implementation of a programming language based on OCL can be greatly 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 [16], the OCL Environment (OCLE) [8], the Eclipse Model Development Tools (MDT) [28], KMF [2], the Octopus tool [21], RoclET [17], Eye OCL [10], HOL-OCL [4], and the UML-based Specification Environment (USE) [13].

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 [27, 7]), expression analysis (e. g., in [12]), and satisfiability checking (e.g., in [6, 11, 23, 3]), theorem proving (e. g., in [4, 25]), and an interoperability benchmark [14].

Thus, there are strong arguments for reusing OCL in model-based programming languages. However, as we will point out, we have to be careful in the definition of an OCL-based programming language: OCL has to be embedded in a non-intrusive manner when we want to take advantage of the aforementioned profits.

2.1. Common Problems

A common problem of the aforementioned OCL-based programming languages is an unresolved conflict between the objective to reuse the side-effect free language OCL on the one hand, and the objective to express state changes on the other. To clarify the first point, we cite the introduction of the OCL specification [38, p. 5]:

> OCL is a pure specification language; therefore, an OCL expression is guaranteed to be without side-effects. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model.

In the following we now regard the conflict in more details. We will refer mainly to ImperativeOCL, but we will also give remarks regarding the other languages mentioned above.

2.1.1. Statements

In imperative programming languages we often refer to their smallest standalone elements as statements. The effect of such a statement is determined by its effect on the environment. For languages that operate on object models, the environment contains at least the available objects, their properties and the links among them, as well as typically some variable assignments.

We can formally describe the semantics of an imperative language in a denotational way in terms of an interpretation function $I[\![\, s \,]\!]$ for each statement s. If we let *env* denote the environment in which we execute s (with everything in it that we need to describe a model instance), the interpretation of a statement $I[\![\, s \,]\!](\mathit{env})$ will be a new environment env'. In contrast, we would typically define the interpretation $I[\![\, e \,]\!](\mathit{env})$ of an expression e (without side-effects) as a value of the domain of the expression type (e.g., a number, a string value, an object).

In many popular languages the distinction is not that clear in general. For example, the assignment `x = y` in Java and C has both an interpretation as a state change and as a value. In this case, we would need to capture the interpretation $I[\![\, s \,]\!](\mathit{env})$ as a tuple (env', y) of a new environment and a value. However, for a non-intrusive 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 semantics 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 imperative *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 combined 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 $I[\![\, e_1 \text{ and } e_2 \,]\!](\mathit{env})$:

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

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

   where $(\mathit{env}', y) = I[\![\, e_1 \,]\!](\mathit{env})$. 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):

$$I\llbracket\, e_1 \text{ and } e_2 \,\rrbracket(env) = \begin{cases} (env'', \text{true}) & \text{if } y_1 = \text{true} \wedge y_2 = \text{true} \\ (env'', \text{false}) & \text{otherwise} \end{cases}$$

where $(env', y_1) = I\llbracket\, e_1 \,\rrbracket(env)$ and $(env'', y_2) = I\llbracket\, e_2 \,\rrbracket(env')$. 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 as a side-effect free language 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, for example, the commutativity of operators. Of course all these little 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.* Similar amalgamations exist in fALF [35], OCL4X [18], and EOL [22][1]. Recall the citation from the OCL specification in the beginning of this section. In our understanding, the change in the interpretation of expressions breaks a fundamental property of the *OclExpression* class. Thus 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 can no 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 safe reuse of OCL. Figure 1 depicts this requirement on the level of the language metamodels. Notice that, in general, an imperative programming language might also add further kinds of pure expressions that are not OCL. However, these expressions must not occur as OCL expressions.

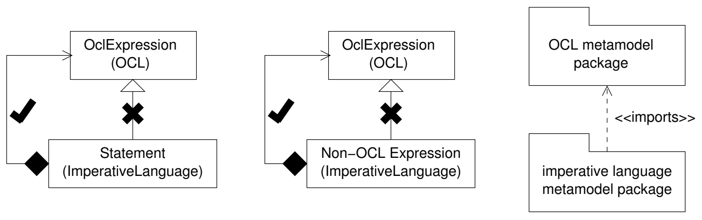

OclExpression (OCL)
Statement (ImperativeLanguage)
OclExpression (OCL)
Non−OCL Expression (ImperativeLanguage)
OCL metamodel package
<<imports>>
imperative language metamodel package

Figure 1: A safe embedding of OCL

2.1.2. Operations with Side-Effects

The invocation 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, the interpretation of an operation with side-effects yields a new state (and possibly a value, too). For similar reasons as before, we should not allow the second one to occur in OCL expressions.

In order not to stretch short-circuit evaluation or commutativity of binary operations for the explanation again, we take a look on the *let* expression in OCL. This expression introduces an intermediate results as a variable. As expected for a logic-like language, the following important equivalence rule holds for OCL:

[1]While EOL explicitly states a loose kind of reuse of OCL in terms of 'being inspired by OCL', the other languages claim a direct reuse of OCL.

$$I[\![\, \text{let } v : T = e_1 \text{ in } e_2 \,]\!](env) = I[\![\, e_2 \,]\!]\big(env[v \mapsto I[\![\, e_1 \,]\!](env)]\big)$$

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'.

```
def: newWorker(firstName : String, lastName : String):Person =
  w := new Worker;
  w.firstName := firstName;
  w.lastName := lastName;
  return w
```

Obviously, the interpretation of

```
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, such problematic situations can be constructed in several ways if we allow operations with side-effects in OCL expressions. For example, for ATL this regards in particular the invocation of lazy rules and called rules from within OCL expressions. Similar examples exists for the other aforementioned languages.

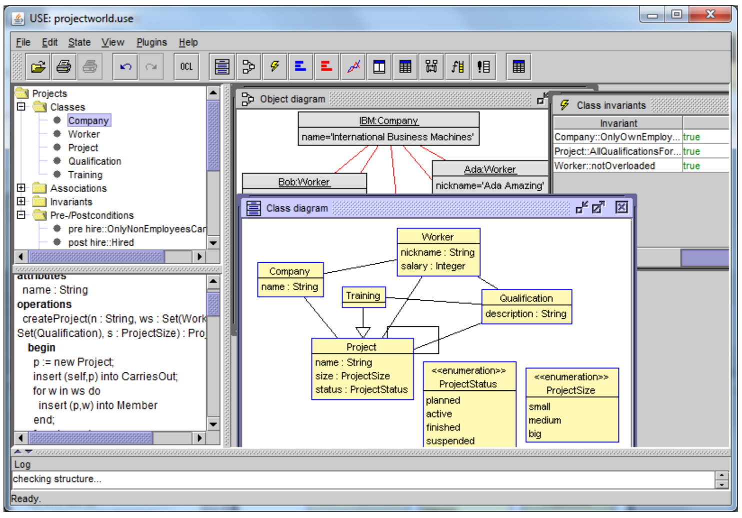
However, for a non-intrusive reuse of OCL, we require a distinction between query operations and operations with side-effects. Within OCL expressions, only query operations must be used. Otherwise, we run into the same problems mentioned in Sect. 2.1.1. Consequently, an imperative language must include an explicit and distinct mechanism to invoke operations with side-effects.

3. SOIL and the UML-based Specification Environment

We now introduce the Simple OCL-based Imperative Language SOIL and the USE tool [13] that implements it. The development of SOIL has been driven by two objectives: The first one is to provide an (OCL-based) model programming language for the USE tool to perform model animation. The second objective has been to study how the (existing) language OCL can be embedded in a way that avoids the aforementioned problems, i.e., without changing the language and without changing existing implementations.

USE basically is an interpreter for OCL and a subset of UML. It supports validating and analyzing the model structure (class diagrams and OCL invariants) and the model behavior (operations and pre- and postconditions) by generating instances ('snapshots') of the model and by executing typical operation sequences ('scenarios'). USE has been employed in various case studies and teaching projects, among other places at the MIT, Cambridge, MA, at the University of Edinburgh, Scotland, at the university of Colorado, CO, and the university of Lisbon, Portugal. It comprises both a command line interface and a graphical user interface, and can also be used as a library for UML and OCL. The graphical user interface is depicted in Fig. 2. Since version 2010, SOIL contributes to this context as the model programming language of USE.

While SOIL is a complete programming language and sufficient for our requirements in the model animation and model transformation contexts, it is not a a general purpose language. It has no I/O, no modules, no concurrency, no higher-order features, and no exception handling. In particular, the domain of SOIL programs are model instances, i.e., objects and relations, vs. objects and references in, e.g., Java. Unlike imperative OO programming languages such as Java and C#, object destruction is explicit in SOIL, like a cascading delete in a database, as most of our model manipulations require this characteristic.



```
attributes
 name : String
operations
 createProject(n : String, ws : Set(Work
Set(Qualification), s : ProjectSize) : Proj
  begin
   p := new Project;
   insert (self,p) into CarriesOut;
   for w in ws do
    insert (p,w) into Member
   end;
```

Invariant	
Company::OnlyOwnEmploy...	true
Project::AllQualificationsFor...	true
Worker::notOverloaded	true

```
Log
checking structure...
Ready.
```

Figure 2: Screenshot of the UML-based Specification Environment (USE)

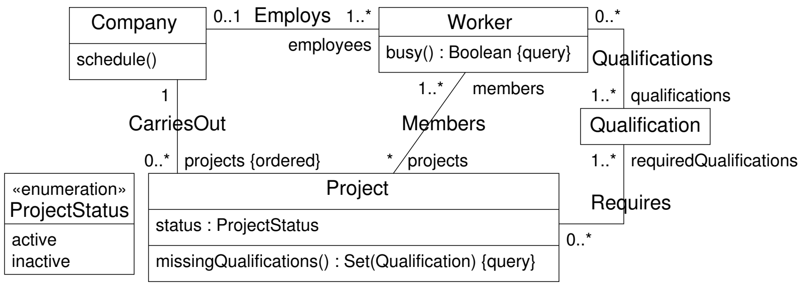


Figure 3: Project World

Apart from its implementation in USE, we see SOIL, however, as a potential alternative to other imperative model transformation languages such as the imperative sub-language of ATL [19], the Epsilon Object Language (EOL) [22], and ImperativeOCL (as part of the QVT specification [36]).

We will employ the following short example to illustrate both USE and SOIL. A bigger, extensive case study conducted using SOIL can be found, for example, in [42]. Consider the class diagram in Fig. 3. 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 many projects as possible).

Some properties of this operation are further specified in a declarative way by OCL postconditions as shown in Listing 1: After scheduling projects, it has to be ensured that no active project lacks 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 in the USE specification file.

Obviously, several implementations of schedule() will fulfill the stated pre- and postconditions. USE allows us to define one using SOIL statements. All SOIL defined operations can be invoked interactively from the command line on the current model instance.

During the animation of the model, all structural and dynamic constraints are checked. In our example, the

```
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: Declarative specification of Company::schedule

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 [31]). However, in our case we are still in the context of the UML object model. USE provides various ways to analyze the instance in case of violated pre- or postconditions.

Listing 2 shows a very simple imperative version of schedule(). We can see that SOIL provides typical control flow constructs (*for*-loop, *if*-statement). Within these statements, OCL expressions are used to query the model (e.g., to describe the range for the iteration and the condition for the *if*-statement). Statements which manipulate the system state are available (in the above example: link insertion and attribute assignment). The semantics of these statements is straightforward. Please notice that we assume that all properties in Fig. 3 are ordered.

```
context Company def: schedule() :=
  begin
    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
  end
```

Listing 2: Operational specification of Company::schedule

Operations with side-effects can be invoked using a specific invocation statement. For example, the following statement uses the scheduling operation defined above. Imperative operations can be invoked recursively.

```
declare c : Company
begin
  c := new Company;
  c.schedule()
end
```

USE processes all aspects of the project world model as introduced in this paper: The static structures can be instantiated (i.e., objects and links can be created). This can happen either manually, using the USE *Generator* (see below), or using SOIL statements. Then, the instantiated system state can be validated against all structural constraints of the model. Regarding the dynamic aspects of a model, any manually provided flow of actions (i.e., a particular sequence of state changes) as well as any execution of a SOIL-defined operation can be validated against the dynamic constraints of the model (i.e., against the pre- and postconditions).

The tool can be employed to validate that our (very simple) implementation of schedule() conforms to the postconditions as follows. For a given initial system state, USE can check if a particular execution of schedule() conforms to its two postconditions. A sufficient coverage of test cases can be provided by means of a surrounding SOIL program, or by employing the *Generator* language [13] of USE (the *Generator* implements a backtracking search to yield valid instances of the model). This kind of validation can be seen as systematic testing, in contrast to a formal verification of correctness.

4. Syntax and Semantics of OCL

Before we define SOIL formally in the next section, we first introduce OCL to the required extent. The normative reference for the language is the OMG specification [38], for an introduction to the language we also refer to [46]. Apart from a metamodel-based, informal description of the language, the specification also provides formal syntax and semantics, based on [43]. We follow that formal description in this section, although we describe the typing rules of OCL explicitly, as in [9, 24]. Without loss of generality for the purpose of this article, we omit some elements of OCL to keep the presentation compact. In particular, we omit tuple types and special operations such as *oclIsKindOf* from the syntax of OCL, and we present only a subset of the interpretation functions for expressions. Adding them does not affect our definition (or implementation) of SOIL.

4.1. Syntax of OCL

The grammar of OCL expressions is as follows:

$$
\begin{array}{lr}
e ::= & \text{(expression)}\\
\quad \text{true} \mid \text{false} \mid 1 \mid 2 \mid 3 \mid \texttt{'abc'} \mid \text{null} \mid \text{invalid} \mid C.\text{allInstances}() \mid \ldots & \text{(constants)}\\
\quad v \mid & \text{(variable)}\\
\quad \text{let } v = e_1 \text{ in } e_2 \mid & \text{(let expr.)}\\
\quad \text{if } e \text{ then } e_1 \text{ else } e_2 \text{ end} \mid & \text{(conditional)}\\
\quad e_1 \rightarrow \text{iterate}(v_1; v_2 = e_2 \mid e_3) \mid & \text{(iterate)}\\
\quad \text{op}(e_1, \ldots, e_n) \mid e_1.\text{op}(e_2, \ldots, e_n) \mid e_1\texttt{->}\text{op}(e_2, \ldots, e_n) \mid e_1 \text{ op } e_2 \mid \text{op}\{e_1, \ldots, e_n\} & \text{(operation call)}
\end{array}
$$

For the basic types (Boolean, Integer, String, ...), corresponding constants can be written as usual. A special constant *C.allInstances* is available that yields the set of objects of this class in the current system state. In addition to the basic types, all classes of the underlying structural model (say, a class diagram) are available as object types. Objects can be, like all primitive values, referenced by variables. A variable *self* is often used to provide the context object for an OCL expression (e.g., when defining an invariant constraint). A *let* expression is provided to introduce a value as a variable in an expression. Condition evaluation is provided by an *if* expression. The *iterate* expression is included here as the most general form of collection operations to iteratively unfold an expression of an expression. Finally, various operations (without side-effects) are provided for all types, including the usual operations of the basic types (explained below), operations on collection types, and several pre-defined operations on all object types, but also user-defined query operations that are defined for the object types. In particular, a query operation *op* for an object type can be defined by an OCL expression e as follows (when p_1 is omitted, *self* is used):

$$\text{context } [p_1 :]\ t_1 :: op(p_2 : t_2, \ldots p_n : t_n) : t \text{ body: } e$$

To describe the syntax and semantics of OCL expressions more precisely, we must first describe the underlying object model $\mathcal{M}$. It is defined in [38, 43] as follows.

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}, \textsc{Op}, \textsc{Assoc}, \text{associates}, \text{roles}, \text{multiplicities}, \prec)$$

$$\text{(T-Basic)}\,\frac{t \in \mathit{BTypes}}{\Gamma \vdash t} \qquad \text{(T-Class)}\,\frac{c \in \textsc{Class}}{\Gamma \vdash c} \qquad \text{(T-Special)}\,\frac{t \in \mathit{STypes}}{\Gamma \vdash t}$$

$$\text{(T-Coll)}\,\frac{\Gamma \vdash t \qquad \mathit{col} \in \mathit{CTypes}}{\Gamma \vdash \mathit{col}(t)} \qquad \text{(T-Sub-Class)}\,\frac{c_1, c_2 \in \textsc{Class} \qquad c_1 \prec c_2}{\Gamma \vdash c_1 <: c_2}$$

$$\text{(T-Sub-Col-1)}\,\frac{\Gamma \vdash \mathit{col}(t) \qquad \mathit{col} \in \mathit{CTypes}}{\Gamma \vdash \mathit{col}(t) <: \text{Collection}(t)} \qquad \text{(T-Sub-Col-2)}\,\frac{\mathit{col} \in \mathit{CTypes} \qquad \Gamma \vdash t <: t'}{\Gamma \vdash \mathit{col}(t) <: \mathit{col}(t')}$$

$$\text{(T-Sub-Trans)}\,\frac{\Gamma \vdash t_1 <: t_2 \qquad \Gamma \vdash t_2 <: t_3}{\Gamma \vdash t_1 <: t_3} \qquad \text{(T-Sub-Refl)}\,\frac{\Gamma \vdash t}{\Gamma \vdash t <: t} \qquad \text{(T-Sub-Any)}\,\frac{\Gamma \vdash t}{\Gamma \vdash t <: \text{OclAny}}$$

$$\text{(T-Sub-Void)}\,\frac{\Gamma \vdash t \qquad t \neq \text{OclInvalid}}{\Gamma \vdash \text{OclVoid} <: t} \qquad \text{(T-Sub-Invalid)}\,\frac{\Gamma \vdash t}{\Gamma \vdash \text{OclInvalid} <: t}$$

Figure 4: Inference rules for types. *BTypes* denotes $\{\text{UnlimitedNatural}, \text{String}, \text{Boolean}, \text{String}, \text{Real}\}$, *STypes* denotes $\{\text{OclAny}, \text{OclVoid}, \text{OclInvalid}\}$, and *CTypes* denotes $\{\text{Collection}, \text{Set}, \text{Bag}, \text{Sequence}, \text{OrderedSet}\}$.

This structure captures 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. We will only explain in more detail the elements that are required for the purpose of our work, and refer to [38, Appendix A] and [43] for further reading.

Given $\mathcal{M}$, the set $\text{types}(\mathcal{M})$ denotes the set of types over $\mathcal{M}$ that can be inferred by the rules as shown in Fig. 4. Summarized, the set of types over a model comprises the pre-defined basic types, the special types OclVoid, and OclInvalid, all class types of $\mathcal{M}$, and all collection types that can be constructed over element types. Because the collection types are template types, $\text{types}(\mathcal{M})$ is infinite. The abstract type Collection is a supertype of all other collection types.

To define the typing rules for OCL expressions, we further need to assume a set $\mathit{Op}_{\text{qry}}(\mathcal{M})$ of all query operations over $\mathcal{M}$. This set comprises in particular all constant symbols, all pre-defined operations on basic types, all navigation operations (dependent on the associations in $\mathcal{M}$), all attribute accessors, and all user-defined class operations (dependent on Op). These are explained in more detail in the next subsection. Notice that we, following the formalization of [38, 43], require that all operations can be statically and unambiguously determined by their name and their parameter types, i.e., we do not support overriding.

Based on $\mathcal{M}$ and $\mathit{Op}_{\text{qry}}(\mathcal{M})$, the typing rules for OCL are given in Fig. 5, where the typing environment comprises a set of variable declarations of the shape $v : t$, where v is a variable name and $t \in \text{types}(\mathcal{M})$. The inference rules for OCL expressions were presented in a similar fashion in [9]. We want to emphasize that (E-Call) allows only (side-effect free) query operations to be used in OCL expressions.

4.2. Semantics of OCL

To describe the semantics of OCL, we need an interpretation of the types over $\mathcal{M}$, an instance σ of $\mathcal{M}$, and the interpretations of query operations over $\mathcal{M}$. We present these definitions, which are not novel, only to the extent that is required for the scope of this article. We refer again to [38, 43] for further details about the semantics of OCL.

1. We require an interpretation $I(t)$ for each type t in $\text{types}(\mathcal{M})$. In particular, $t_1 <: t_2$ implies $I(t_1) \subset I(t_2)$. All interpretations of types include the value ϵ (*null*) and the value $\perp$ (*invalid*). The first one is used to denote an unknown fact, the second one denotes an error (e.g., division by zero). For example, the interpretation $I(\text{Integer})$ of the type *Integer* is $\mathbb{Z} \cup \{\epsilon, \perp\}$.
2. An instance σ of $\mathcal{M}$ describes a structure of objects, their attribute values, and the links between the objects (say, an object diagram). We let $\sigma(c)$ denote the set of instances of c in σ for each c in Class.

$$\text{(E-Sub)}\frac{\Gamma \vdash e : t_1 \qquad \Gamma \vdash t_1 <: t_2}{\Gamma \vdash e : t_2} \qquad \text{(E-Var)}\frac{v : t \in \text{dom}(\Gamma)}{\Gamma \vdash v : t} \qquad \text{(E-Let)}\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, v : t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } v = e_1 \text{ in } e_2 : t_2}$$

$$\text{(E-Call)}\frac{op_{\text{qry}} : t_1 \times \cdots \times t_n \to t \in Op_{\text{qry}} \qquad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_n : t_n}{\Gamma \vdash op_{\text{qry}}(e_1, \ldots, e_n) : t}$$

$$\text{(E-Cond)}\frac{\Gamma \vdash e_1 : \text{Boolean} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\text{if } e \text{ then } e_1 \text{ else } e_2 \text{ end} : t}$$

$$\text{(E-ColLit)}\frac{\Gamma \vdash e_1 : t \ldots \Gamma \vdash e_n : t \qquad col \in \{\text{Set}, \text{Bag}, \text{Sequence}, \text{OrderedSet}\}}{\Gamma \vdash col\{e_1, \ldots, e_n\} : col(t)}$$

$$\text{(E-Iterate)}\frac{\Gamma \vdash e_1 : \text{Collection}(t) \qquad \Gamma \vdash e_2 : t_2 \qquad \Gamma \vdash e_3 : t_2}{\Gamma \vdash e_1 \to \text{iterate}(v_1; v_2 = e_2 | e_3) : t_2}$$

Figure 5: Inference rules for OCL expressions

We let $\sigma(att)$ denote the set of attribute assignments $o \mapsto x$ with o being in $\sigma(c)$ and $x \in I(t)$ for each $att : c \to t \in \text{ATT}$. We let $\sigma(a)$ denote the set of links of the shape $\langle o_1, \ldots, o_n \rangle$ for each association a in ASSOC (with n according to the multiplicity of a). Finally, we let INSTANCES denote the set of all instances of $\mathcal{M}$.

3. We require an interpretation $I(op_{\text{qry}})$ for each query operation $op_{\text{qry}} : t_1 \times \cdots \times t_n \to t$ in $Op_{\text{qry}}(\mathcal{M})$ that assigns a function $\text{INSTANCES} \times I(t_1) \times \cdots \times I(t_n) \to I(t)$ to it[2] (and for parameter-less query operations a function $\text{INSTANCES} \to I(t)$ respectively).

Having the interprations $I(t)$ of types, the set INSTANCES of all instances of $\mathcal{M}$, and the interpretations $I(op_{\text{qry}})$ of query operations, we can express the interpretation $I[\![e]\!]$ of an expression e by a function $I[\![e]\!]$ which assigns a value to each pair $env = (\sigma, \beta)$ of an instance σ of $\mathcal{M}$ and a variable assignment $\beta : \text{Var}_t \to I(t)$.

Let Env_{qry} be the set of environments $env = (\sigma, \beta)$. The semantics of an expression $e : t$ is a function $I[\![e]\!] : \text{Env}_{\text{qry}} \to I(t)$. The interpretation function is made total by including the *invalid* value $\perp$ in the interpretation of all types.

The full denotational semantics of OCL is described in [38, A.5], so we only show the interpretations of some selected expressions (quoting the OCL spec. up to renaming of symbols). For atomic expressions, the interpretation of constants (considered as query operations without parameters) is given by

$$I[\![op]\!] = I(op) \qquad \text{resp.} \qquad I[\![op()]\!] = I(op).$$

The OCL spec. defines several operations that an object model $\mathcal{M}$ generates. In particular, $I(1) = 1$, $I(2) = 2$, etc., and $I(\text{invalid}) := \perp$ [3] and $I(\text{null}) := \epsilon$. Also, for each class c in CLASS, there is an operation $I(c.\text{allInstances}) := \sigma(c)$ yielding all instances of an object type.

Query operations with parameters can be invoked in several syntactic ways (infix, prefix, or postfix, with a dot or with an arrow), but their interpretation is eventually given in the same way:

$$I[\![op_{\text{qry}}(e_1, \ldots, e_n)]\!](env) = I(op_{\text{qry}})(\sigma)(I[\![e_1]\!](env), \ldots, I[\![e_n]\!](env)).$$

[2]Here we deviate from [38, 43, A.4.8] and include INSTANCES to the signature of the interpretation function, as several functions require the state in their definition (e.g, navigation operations). It might be assumed implicitly in [38, 43].

[3]Here we deviate slightly from the OCL spec [38], which uses the literal *undefined* here. We assume that this is a mistake in the spec.

An object model $\mathcal{M}$ generates several query operations, in particular the operations for the primitive data types (e.g., +, -, *, ...), accessor operations for all attributes and all association ends, and collection operations (e.g., size and the constructors collection literals). Except for the logical connectives, all query operations are defined *strict*, in the sense that they evaluate to $\perp$ if any of their parameters is $\perp$. For the logical connectives, e.g., (true or $\perp$) = true holds. In addition the query operations generated by $\mathcal{M}$, $I(Op_{\mathrm{qry}})$ also contains interpretations for all user defined query operations (which can be specified by an OCL expression, see [38, A.5.1.2]).

Variables (which have to bound in some outer context) generate expressions, too.

$$I[\![\, v \,]\!](env) := \beta(v)$$

For example, the let expression can introduce variables.

$$I[\![\, \text{let } v = e_1 \text{ in } e_2 \,]\!](env) := I[\![\, e_2 \,]\!]\big((\sigma, \beta[v \mapsto I[\![\, e_1 \,]\!](env)])\big)$$

The conditional is defined strict on its condition. When e_1 evaluates to $\perp$ or ϵ, the whole expression evaluates to $\perp$. It is non-strict on its branches, i.e., it does not propagate an invalid value $\perp$ for the branch that is not selected:

$$I[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ end} \,]\!](env) := \begin{cases} I[\![\, e_2 \,]\!](env) & I[\![\, e_1 \,]\!](env) = \text{true}, \\ I[\![\, e_3 \,]\!](env) & I[\![\, e_2 \,]\!](env) = \text{false}, \\ \perp & \text{otherwise.} \end{cases}$$

As initially said, there are more kinds of expressions that we do not cover here, such as the *iterate* expression (which provides a folding function that can be used to implement various operations on collections like *forAll*, *exists*, *collect*). Also, some operations are available that take types are arguments (and are not provided in $I(Op_{\mathrm{qry}})$), like oclIsTypeOf(t), which performs a type test, or oclAsType(t), which performs a type down-cast.

4.3. Postconditions Constraints

In the context of postcondition constraints for an operation, OCL provides an extended set of expression, that furthermore introduces an *@pre* version of all operations, referring to the *pre-state* of the constrained operation, and an additional operation *oclIsNew*, that can test whether an object has been freshly created by the constrained operation.

In this context, the semantics of an expression e of type t is given by an interpretation function $I[\![\, e \,]\!] : \mathrm{Env}_{\mathrm{qry}} \times \mathrm{Env}_{\mathrm{qry}} \to I(t)$ that evaluates an expression given a pre-state σ_{pre} and a post-state σ_{post}. In our language SOIL (introduced right below), we do not require this form of evaluation. OCL postconditions can, however, be used to constrain operations that are imperatively defined by SOIL statements (USE, for example, supports this).

5. Syntax and Semantics of SOIL

We now provide precise semantics for SOIL. We follow the same structure as we did in the previous section for OCL. First, in Sect. 5.1, we explain the grammar and the typing rules. Then, in Sect 5.2, we define the semantics of statements by an interpretation function.

5.1. *Syntax of SOIL*

The grammar of SOIL, which is a grammar of statements, is defined as follows, where e denotes an OCL expression and v denotes a variable name, as in the previous section.

$s ::=$	(statement)
$v :=$ new c |	(1. – object creation)
destroy e |	(2. – object destruction)
insert $(e_1, \ldots, e_n)$ into a |	(3. – link insertion)
delete $(e_1, \ldots, e_n)$ from a |	(4. – link deletion)
$e_1.a := e_2$ |	(5. – attribute assignment)
$v := e$ |	(6. – variable assignment)
$e_1.op(e_2, \ldots, e_n)$ |	(7. – operation call)
$v := e_1.op(e_2, \ldots, e_n)$ |	(8. – operation call with result)
[declare $v_1 : t_1, \ldots, v_n : t_n$] [begin] $s_1; \ldots; s_n$ [end] |	(9. – block of statements)
if e then s_1 [else s_2] end |	(10. – conditional execution)
for v in e do s end	(11. – iteration)

Statement (1)–(5) are the basic state manipulation statements. They provide a means to modify a model instance, i.e., to create objects (storing the newly created object in a variable), to destroy an object, to insert and delete links from associations, and to modify an attribute value of an object. The resp. objects and values in these statements and the remaining statements are specified by OCL expressions, as defined in the previous section (the grammar of SOIL statements is dependent on the grammar of OCL expressions). Statement (6) is the variable assignment. It captures the result of an expression and stores it in a variable, making it available in subsequent statements of the same scope. (7) and (8) regard the invocation of an expression with (possible) side-effects, where (8) regards operations that have a return value. (9) groups a sequence of statements into a block, optionally introducing local variables for that block. (10) and (11) provide conditional execution and an iterate loop. When a block of statements without local variables is used in these statements, the *begin* and *end* keywords of the block can be omitted for brevity.

For the formalization of SOIL statements over a model $\mathcal{M}$, we require again the set $Op_{\text{qry}}(\mathcal{M})$ of all query operations, plus a new set $Op_{\text{imp}}(\mathcal{M})$ of all operations with side-effects over $\mathcal{M}$. Notice that we assume (as for OCL) that operations can be statically and unambiguously inferred from the operation name and the parameter types, i.e., we do not support overriding. The consequence is that we can treat operations as functions. Although overriding is generally desirable for object-oriented programming languages, we do not regard it as critical for the limited application scope of SOIL, and it would add significantly to the complexity of the language syntax and semantics. We refer, for example, to [1] for an in-depth treatment for the formalization of object-oriented imperative languages with overriding.

The typing rules for SOIL are depicted in Fig. 6. As for the type inference rules for OCL, we again assume Γ to be the set of declared variables $v : t$. Notice furthermore that some rules that infer statements (Fig. 6) require a judgment $\Gamma \vdash e$ for an expression e in their premises.

5.1.1. *Defining Operations by SOIL statements*

Operations with side-effects are specified by a SOIL statement:

$$\text{context } [p_1 :]\ t_1 :: op(p_2 : t_2, \ldots p_n : t_n)\ [\ : t\ ] \text{ body: } s$$

Like for contextual definitions in OCL, the parameter p_1 can be omitted; in that case it is assumed to be *self*. We formalize the above notation as follows: The mapping impopdef assigns to each operation $op : p_1 : t_1, \ldots, p_n : t_n$ (resp., $op_i : p_1 : t_1, \ldots, p_n : t_n \to t$) in Op_{imp} a statement s such that $\{p_1 : t_1, \ldots, p_n : t_n\} \vdash s$ (resp., $\{p_1 : t_1, \ldots, p_n : t_n, \text{result} : t\} \vdash s$).

$$(\text{S-Var})\,\frac{v : t \in \Gamma \qquad \Gamma \vdash e : t}{\Gamma \vdash v := e}$$

$$(\text{S-Att})\,\frac{c \in \textsc{Class} \qquad a : c \to t \in \textsc{Att} \qquad \Gamma \vdash e_1 : c \qquad \Gamma \vdash e_2 : t}{e_1.a := e_2}$$

$$(\text{S-LinkIns})\,\frac{a \in \textsc{Assoc} \qquad \text{associates}(a) = \langle c_1, \ldots, c_n \rangle \qquad \Gamma \vdash e_1 : c_1 \ldots \Gamma \vdash e_n : c_n}{\Gamma \vdash \text{insert } (e_1, \ldots, e_n) \text{ into } a}$$

$$(\text{S-LinkDel})\,\frac{a \in \textsc{Assoc} \qquad \text{associates}(a) = \langle c_1, \ldots, c_n \rangle \qquad \Gamma \vdash e_1 : c_1 \ldots \Gamma \vdash e_n : c_n}{\Gamma \vdash \text{delete } (e_1, \ldots, e_n) \text{ from } a}$$

$$(\text{S-Call1})\,\frac{op : t_1 \times \cdots \times t_n \in Op_{\text{imp}} \qquad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_n : t_n}{\Gamma \vdash op(e_1, \ldots, e_n)}$$

$$(\text{S-Call2})\,\frac{op : t_1 \times \cdots \times t_n \to t \in Op_{\text{imp}} \qquad \Gamma \vdash e_1 : t_1 \ldots \Gamma \vdash e_n : t_n \qquad v : t' \in \Gamma \qquad \Gamma \vdash t <: t'}{\Gamma \vdash v := op(e_1, \ldots, e_n)}$$

$$(\text{S-Block})\,\frac{\{v_1, \ldots, v_m\} \cap \text{dom}(\Gamma) = \emptyset \qquad \Gamma, v_1 : t_1, \ldots, v_m : t_m \vdash s_1 \ \ldots \ \Gamma, v_1 : t_1, \ldots, v_m : t_m \vdash s_n}{\Gamma \vdash \ \text{declare } v_1 : t_1, \ldots, v_m : t_m \text{ begin } s_1; \ldots; s_n \text{ end}}$$

$$(\text{S-Iter})\,\frac{v \notin \text{dom}(\Gamma) \qquad \Gamma \vdash e : \text{Sequence}(t) \text{ or } \Gamma \vdash e : \text{OrderedSet}(t) \qquad \Gamma, v : t \vdash s}{\Gamma \vdash \text{for } v \text{ in } e \text{ do } s \text{ end}}$$

$$(\text{S-Cond1})\,\frac{\Gamma \vdash e : \text{Boolean} \qquad \Gamma \vdash s}{\Gamma \vdash \text{if } e \text{ then } s \text{ end}} \qquad (\text{S-Cond2})\,\frac{\Gamma \vdash e : \text{Boolean} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end}}$$

Figure 6: Type rules for the imperative language SOIL

5.2. Semantics of SOIL

To describe the semantics of SOIL statements over $\mathcal{M}$, we assume *Env* to contain all pairs $env = (\sigma, \beta)$ of an instance σ of $\mathcal{M}$ and list of variable assignments $\beta = \langle B_1, \ldots, B_n \rangle$, plus err_{imp} to denote a runtime error. We lift *Env* to $\lfloor \text{Env} \rfloor = Env \cup \{\bot_{\text{imp}}\}$, where $\bot_{\text{imp}}$ denotes an undefined environment due to divergence.

Furthermore, we assume *FEnv* to be the set of all environments $\varphi = (\varphi_1, \ldots, \varphi_n)$ of functions $\lfloor \text{Env} \rfloor \times I(t_1) \times \cdots \times I(t_{nk}) \to \lfloor \text{Env} \rfloor$ ($\lfloor \text{Env} \rfloor \times I(t_1) \times \cdots \times I(t_{nk}) \to (\lfloor \text{Env} \rfloor \times I(t))$) corresponding to each operation op_i in Op_{imp} without (with) return value. The interpretation of operation calls characterizes φ recursively, and we use its least fixed point δ (shown below) to define the semantics as

$$I[\![\, s \,]\!](env) = I[\![\, s \,]\!](\delta)(env).$$

We make use of the following auxiliary definitions.

- We use top(env) to denote that most recent 'frame' of variables in a (defined, non-errorroneous) environment env, $\text{top}(\sigma, \langle B_1, \ldots, B_n \rangle) := B_n$. In particular, we can use it to evaluate an OCL expression in the most recent frame of env as $I[\![\, e \,]\!](\text{top}(env))$.
- We use push(env) and pop(env) to denote adding and removing a frame to (from) env. For $env = (\sigma, \langle B_1, \ldots, B_n \rangle)$, we have $\text{push}(env) = (\sigma, \langle B_1, \ldots, B_n, \emptyset \rangle)$ and $\text{pop}(env) = (\sigma, \langle B_1, \ldots, B_{n-1} \rangle)$. We have $\text{push}(\bot_{\text{imp}}) = \bot_{\text{imp}}$ and $\text{push}(\text{err}_{\text{imp}}) = \text{err}_{\text{imp}}$ ($\text{pop}(\bot_{\text{imp}})$ and $\text{pop}(\text{err}_{\text{imp}})$ resp.).
- We use $env[v \mapsto x]$ to denote an assignment of a value x to v in env. We define it as $(\bot_{\text{imp}})[v \mapsto x] := \bot_{\text{imp}}$, $(\text{err}_{\text{imp}})[v \mapsto x] := \text{err}_{\text{imp}}$, and $(\sigma, \langle B_1, \ldots, B_n \rangle)[v \mapsto x] := (\sigma, \langle B_1, \ldots, B_n[v \mapsto x] \rangle)$ otherwise.

Having these preliminaries, we can describe the semantics of a statement s by an interpretation function $I[\![\, s \,]\!] : FEnv \to (\lfloor \text{Env} \rfloor \to \lfloor \text{Env} \rfloor)$ as follows:

5.2.1. Propagation

We have $I[\![\, s \,]\!](\varphi)(\mathrm{err}_{\mathrm{imp}}) := \mathrm{err}_{\mathrm{imp}}$ and $I[\![\, s \,]\!](\varphi)(\bot_{\mathrm{imp}}) := \bot_{\mathrm{imp}}$.

5.2.2. Interpretation of basic statements

1. For $\mathit{env} = (\sigma, \beta)$ and a fresh object obj that is not in $\sigma(c)$ and $\beta = \langle B_1, \ldots, B_n \rangle$,
   $I[\![\, v := \mathrm{new}\ c \,]\!](\varphi)(\mathit{env}) := \mathit{env}'$
   where $\mathit{env}' = (\sigma', \beta')$ with $\sigma' := \sigma$ except $\sigma'(c') := \sigma(c') \cup \{\mathit{obj}\}$ for each $c' \in \textsc{Class}$ with $c \prec c'$ and $\beta' := \langle B_1, \ldots, B_n[v \mapsto \mathit{obj}] \}$.
2. For $\mathit{env} = (\sigma, \beta)$ and $y = I[\![\, e \,]\!]\big(\mathrm{top}(\mathit{env})\big)$,
   $$I[\![\, \mathrm{destroy}\ e \,]\!](\varphi)(\mathit{env}) := \begin{cases} \mathit{env}' & \text{if } y \notin \{\bot_{\mathrm{ocl}}, \epsilon\} \\ \mathrm{err}_{\mathrm{imp}} & \text{otherwise.} \end{cases}$$
   where $\mathit{env}' = (\sigma', \beta')$ is equal to env except that all links containing y are removed from $\sigma(a)$ (for every $a \in \textsc{Assoc}$) and that all assignments $x \mapsto z$ in β and $\sigma(\mathit{att})$ (for every $\mathit{att} \in \textsc{Att}$) are replaced by $x \mapsto \epsilon$ when z is equal to y or contains y as a component (for tuples and collections).
3. For $\mathit{env} = (\sigma, \beta)$ and $x_1, \ldots, x_n = I[\![\, e_1 \,]\!]\big(\mathrm{top}(\mathit{env})\big), \ldots, I[\![\, e_n \,]\!]\big(\mathrm{top}(\mathit{env})\big)$,
   $$I[\![\, \mathrm{insert}(e_1, \ldots, e_n)\ \mathrm{into}\ a \,]\!](\varphi)(\mathit{env}) := \begin{cases} (\sigma', \beta) & \text{if } x_i \notin \{\bot_{\mathrm{ocl}}, \epsilon\} \text{ for each } i \text{ with } 1 \le i \le n, \\ \mathrm{err}_{\mathrm{imp}} & \text{otherwise} \end{cases}$$
   where $\sigma' := \sigma$ except $\sigma'(a) := \sigma(a) \cup \big\{(x_1, \ldots, x_n)\big\}$; and
   $$I[\![\, \mathrm{delete}(e_1, \ldots, e_n)\ \mathrm{into}\ a \,]\!](\varphi)(\mathit{env}) := \begin{cases} (\sigma', \beta) & \text{if } x_i \notin \{\bot_{\mathrm{ocl}}, \epsilon\} \text{ for each } i \text{ with } 1 \le i \le n, \\ \mathrm{err}_{\mathrm{imp}} & \text{otherwise} \end{cases}$$
   where $\sigma' := \sigma$ except $\sigma'(a) := \sigma(a) - \big\{(x_1, \ldots, x_n)\big\}$.
4. For $\mathit{env} = (\sigma, \beta)$, $x = I[\![\, e_1 \,]\!]\big(\mathrm{top}(\mathit{env})\big)$, and $y = I[\![\, e_2 \,]\!]\big(\mathrm{top}(\mathit{env})\big)$,
   $$I[\![\, e_1.\mathit{att} := e_2 \,]\!](\varphi)(\mathit{env}) := \begin{cases} (\sigma', \beta) & \text{if } x \notin \{\bot_{\mathrm{ocl}}, \epsilon\} \text{ and } y \neq \bot_{\mathrm{ocl}} \\ \mathrm{err}_{\mathrm{imp}} & \text{otherwise} \end{cases}$$
   where $\sigma' := \sigma$ except $\sigma'(\mathit{att}) := \sigma(\mathit{att})\big[x \mapsto y\big]$
5. For $\mathit{env} = (\sigma, \beta)$ and $x = I[\![\, e \,]\!]\big(\mathrm{top}(\mathit{env})\big)$,
   $$I[\![\, v := e \,]\!](\varphi)(\mathit{env}) := \begin{cases} (\mathit{env}[v \mapsto x]\rangle) & x \neq \bot_{\mathrm{ocl}} \\ \mathrm{err}_{\mathrm{imp}} & \text{otherwise} \end{cases}$$

The semantics of the atomic statements is straightforward in general, but we want to comment on some aspects. First, as a general remark, we want to emphasize how the definition of $I[\![\, s \,]\!]$ is dependent on the definition of $I[\![\, e \,]\!]$. In particular, given a state $\mathit{env} = (\sigma, \beta)$, OCL expressions are always evaluated in the form $I[\![\, e \,]\!](\mathrm{top}(\mathit{env}))$, which is a shortcut for $I[\![\, e \,]\!](\sigma, B_n)$, given that $\beta = \langle B_1, \ldots, B_n \rangle$. The typing rules for the language guarantee that B_n contains all free variables of e.

Second note that the invalid value $\bot_{\mathrm{ocl}}$ of OCL is propagated to $\mathrm{err}_{\mathrm{imp}}$ for all expressions within a statement.

The third remark regard the destruction of objects. As mentioned before, SOIL intentionally has no garbage collection, as garbage collection is not suited for the manipulation of instances, where we take a ‘global’ perspective on objects and cannot assume defined roots that could be used to determine reachability. Therefore, SOIL has an explicit *destroy* statement. It is not sufficient to simply remove an object obj from $\sigma(c)$. We furthermore have to remove all dangling references to obj. That is, we have to remove all links from σ that contain obj in one of its component, and we have to replace all occurrences of obj in attribute assignments and in the variable stack β by the OCL undefined value ϵ. Notice that this is a deep modification of the variable stack, we do not only replace the occurrences in the current frame, but also in previous frames.

5.2.3. Compound statements

Having defined the interpretation of atomic statements, we now define the interpretation of compound statements for $env = (\sigma, \beta)$.

1. For $env = (\sigma, \beta)$,
   $I[\![\, \text{declare } v_1 : t_1, \ldots, v_m : t_m \text{ begin } s_1; \ldots; s_k \text{ end} \,]\!](\varphi)(env) :=$
   $I[\![\, s_k \,]\!](\varphi)\Big(\ldots I[\![\, s_2 \,]\!](\varphi)\big(I[\![\, s_1 \,]\!](\varphi)(env')\big)\Big)$
   where $env' = env[v_1 \mapsto \epsilon, \ldots, v_m \mapsto \epsilon]$.

2. For $env = (\sigma, \beta)$ and $b = I[\![\, e \,]\!]\big(\text{top}(env)\big)$,
   $$I[\![\, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \,]\!](\varphi)(env) := \begin{cases} I[\![\, s_1 \,]\!](\varphi)(env) & \text{if } b = \text{true} \\ I[\![\, s_2 \,]\!](\varphi)(env) & \text{if } b = \text{false or } b = \epsilon \\ \text{err}_{\text{imp}} & \text{otherwise} \end{cases}$$
   and
   $$I[\![\, \text{if } e \text{ then } s_1 \text{ end} \,]\!](\varphi)(env) := \begin{cases} I[\![\, s_1 \,]\!](\varphi)(env) & \text{if } b = \text{true} \\ env & b = \text{false or } b = \epsilon \\ \text{err}_{\text{imp}} & \text{otherwise} \end{cases}$$

3. For $env = (\sigma, \beta)$,
   $$I[\![\, \text{for } v \text{ in } r \text{ do } s \text{ end} \,]\!](\varphi)(env) := \begin{cases} \text{err}_{\text{imp}} & \text{if } I[\![\, r \,]\!]\big(\text{top}(env)\big) \in \{\bot_{\text{ocl}}, \epsilon\} \\ env & \text{if } I[\![\, r \,]\!]\big(\text{top}(env)\big) = \langle\rangle \\ I[\![\, s' \,]\!](\varphi)\big(I[\![\, s \,]\!](\varphi)(env[v \mapsto x_1])\big) & \text{otherwise} \end{cases}$$
   where $I[\![\, r \,]\!]\big(\text{top}(env)\big) = \langle x_1, \ldots, x_n \rangle$ and $s' = \text{for } v \text{ in Sequence}\{x_2, \ldots, x_n\} \text{ do } s \text{ end}$.

The interpretation of compound statements is straightforward again. The block statement assigns the OCL undefined value ϵ for all variable in the *declare* clause and then applies the interpretations of s_i from left to right. The block statement does not introduce a new frame on the variable stack. This is not necessary, as the type system does not allow hiding of variables for simplicity. The same remark holds for the iteration, which introduces the loop variable.

5.2.4. Operation invocations

Next, we define the interpretation of operation calls. Unlike the interpretation of query operations, the interpretation of imperative operations does yield a new state. For the reasons of object destruction, as explained above, the interpretation of operation calls also yields a new variable stack in which references might have been changed to ϵ by the interpretation of the called operation. As for basic statements, operation invocation propagates errorneous or undefined environments.

1. For $env = (\sigma, \beta)$, $op_i : (p_1 : t_1, \ldots, p_n : t_n) \in Op_{\text{imp}}$, and
   $x_1, \ldots, x_n = I[\![\, e_1 \,]\!]\big(\text{top}(env)\big), \ldots, I[\![\, e_n \,]\!]\big(\text{top}(env)\big)$,
   $$I[\![\, e_1.\, op_i(e_2, \ldots, e_n) \,]\!](\varphi)(env) := \begin{cases} \varphi_i(env, x_1, \ldots, x_n) & \text{if } x_1 \neq \epsilon \text{ and } x_i \neq \bot_{\text{ocl}} \\ & \text{ for each } i \text{ with } 1 \leq i \leq n \\ \text{err}_{\text{imp}} & \text{otherwise.} \end{cases}$$
2. For $env = (\sigma, \beta)$, $op_i : (p_1 : t_1, \ldots, p_n : t_n \to t) \in Op_{\text{imp}}$, and
   $x_1, \ldots, x_n = I[\![\, e_1 \,]\!]\big(\text{top}(env)\big), \ldots, I[\![\, e_n \,]\!]\big(\text{top}(env)\big)$,
   $$I[\![\, v := e_1.\, op_i(e_2, \ldots, e_n) \,]\!](env) := \begin{cases} env'[v \mapsto y] & \text{if } x_1 \neq \epsilon \text{ and } x_i \neq \bot_{\text{ocl}} \\ & \text{ for each } i \text{ with } 1 \leq i \leq n \\ \text{err}_{\text{imp}} & \text{otherwise.} \end{cases}$$
   where $(env', y) = \varphi_i(env, x_1, \ldots, x_n)$.

The function environment φ is specified recursively by the statements that are assigned to the operation symbols. Thus, it is determined by the least fixed point δ of the function $F : \mathit{FEnv} \to \mathit{FEnv}$ constructed by tupling with

$$\begin{aligned} F(\varphi) = \big(&\lambda \mathit{env}, x_1, \ldots, x_{1k} \,.\, \mathrm{call}(\mathrm{impopdef}(\mathit{op}_1), \varphi, \mathit{env}, x_1, \ldots, x_{1k}),\\ &\ldots,\\ &\lambda \mathit{env}, x_1, \ldots, x_{nk} \,.\, \mathrm{call}(\mathrm{impopdef}(\mathit{op}_n), \varphi, \mathit{env}, x_1, \ldots, x_{nk}), \end{aligned}$$

using

$$\mathrm{call}(s, \varphi, \mathit{env}, x_1, \ldots, x_k) = \mathrm{pop}\big(I[\![\, s \,]\!](\mathrm{push}(\mathit{env})[p_1 \mapsto x_1, \ldots, p_n \mapsto x_n])\big)$$

for operations without return value and

$$\mathrm{call}(s, \varphi, \mathit{env}, x_1, \ldots, x_k) = (\mathrm{pop}(\mathit{env}'), y)$$

with $\mathit{env}' = I[\![\, s \,]\!](\mathrm{push}(\mathit{env}[p_1 \mapsto x_1, \ldots, p_n \mapsto x_n]))$ and $y = I[\![\, \mathrm{result} \,]\!](\mathrm{top}(\mathit{env}'))$ for operations with return value.

Notice that such a least fixed point exists since $\lfloor \mathrm{Env} \rfloor$ forms a complete partial order (with a flat ordering on *Env* and $\bot_{\mathrm{imp}}$ as least element) and the denotations for statements (given a function environment) are continuous functions [47, pp. 124].

5.3. Remarks

The syntax and semantics of SOIL are consistent to each other in the following sense. We say that an environment $\mathit{env} = (\sigma, \beta)$ is *valid* whenever neither σ nor β contain references to objects that do not exist in σ, and β is non-empty.

Proposition 1. *Let s be a statement with $v_1 : t_1, \ldots, v_n : t_n \vdash s$ and a valid environment env with $\mathrm{top}(\mathit{env})(v_i) \in I(t_i)$ for each i, $1 \leq i \leq n$. Then $I[\![\, s \,]\!](\mathit{env})$ is either a valid environment or $\bot_{imp}$.*

Proof sketch. The above proposition can be proved by structural recursion on the interpretations (we assume the interpretations of $I[\![\, e \,]\!]$ to be type-sound). In particular, we can establish that in each induction step all occurrences of $I[\![\, e \,]\!]$ will always have the required values (with the required types) in the passed variable bindings. □

6. Discussion

In the previous section we have formally defined the imperative language SOIL, which reuses OCL. Let us recall the different layers of reuse:

- The grammar of SOIL statements ($s ::= \ldots$) contains OCL expressions e. However, we did not change the grammar of expressions itself.
- The typing rules for statements refer to the typing rules of expressions, that is, a judgment $\Gamma \vdash e : t$ may occur in the premise of the typing rules for statements, but not in the conclusion. This means, that the typing for OCL expressions is not affected by the typing rules we introduced for SOIL.
- In similar vein, the interpretation $I[\![\, s \,]\!](\mathit{env})$ depends on the interpretation of expressions. More specifically, the definition of $I[\![\, s \,]\!](\mathit{env})$ uses $I[\![\, e \,]\!](\mathrm{top}(\mathit{env}))$ to determine the meaning of an OCL expression in the context of the topmost stack frame.

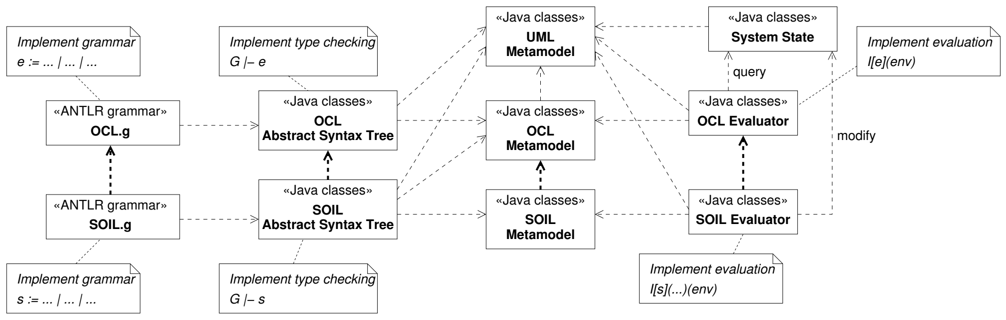


Figure 7: Dependencies between the OCL and SOIL components in USE. The figure shows how the dependency direction SOIL $\rightarrow$ OCL (but not vice versa) is mirrored by the program components of USE.

- Finally, we want to emphasize that OCL expressions may call only query operations (given by the set $Op_{\text{qry}}(\mathcal{M})$), whose interpretation is a value of the return type of that operation, whereas a call statement in SOIL refers to an imperative operation that may have side-effects. The interpretation of these operations is a new environment (and optionally, a value of the result type).

In this sense, our definition of SOIL reflects the message of Fig. 1. In particular, SOIL can be in principle implemented on top of existing OCL implementations, coding only the SOIL part and reusing an existing OCL package or library for the OCL evaluations and the model handling[4]. This can be illustrated using the UML-based specification environment (USE), which we have extended to support SOIL (with USE supporting OCL since many years already). We will discuss the extended version from a user perspective in the next section, but we want to already have a look on the dependencies between its different program components at this point. The components responsible for parsing, type-checking, and evaluating OCL and SOIL are presented in Fig. 7. We can see that they closely correspond to the definitions of the syntax and semantics of the languages. For both languages, a grammar file for the ANTLR parser generator [40] is used to generate a parser that accepts the grammars ($e ::= \dots$, $s ::= \dots$) described before. While parsing, an abstract syntax tree (AST) is created that, after the parsing, performs the type-checking, as described in Figs. 5 and 6. Finally, a type-checked instance of the OCL (resp. SOIL) meta-model is instantiated from the AST. An evaluator class is provided for both the evaluation of expressions and the execution of statements. One can see clearly how the dependency SOIL $\rightarrow$ OCL (but not OCL $\rightarrow$ SOIL) of the formal definition is mirrored one-to-one in the implementation of USE. We did not have to change the components for OCL in order to implement SOIL.

We expect that SOIL can be implemented similary on top of other OCL tools and libraries such as EMF OCL or the Dresden OCL toolkit.

To our knowledge, OCL has not been embedded in this fashion in other programming (or transformation) languages before. In particular, the embeddings of OCL in ImperativeOCL [36], ATL [19, 20], EOL [22], OCL4X [18], and fALF [35] all suffer from the problems illustrated in the beginning.

6.1. Drawbacks of a our Embedding

Having explained the advantages of a safe, non-intrusive embedding of OCL, we will now also regard some drawbacks that arise from it.

In general, apart from the syntactical differences, languages that reuse OCL in the way SOIL does can express imperative programs in a similar way as languages like ImperativeOCL, that embed OCL in an

[4]Of course, up to the usual technical obstacles not related to OCL or SOIL. . .

intrusive way. There are, however, kinds of statements that cannot be translated one-to-one. Specifically, these statements are statements that contain expressions that contain statements. Constructions such as the following ImperativeOCL compute expression

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

cannot be expressed as one atomic statement in SOIL. Such an amalgamation of expressions that contain statements has to be resolved in several steps:

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

Notice that these cases of amalgamations also include invocations of non-query (i.e., side effected) operations from OCL expressions (e.g., the invocation of *lazy rules* and *called rules* in the OCL expressions of ATL). This can be illustrated assuming f and g to be operations with side-effects that furthermore yield integer values; the following ImperativeOCL expression

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

has to be rewritten in SOIL to

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

Of course an 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. 1 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. 2. One might regard 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 would end up doubling the OCL definition and effectively not *reusing* OCL anymore. If we only allow certain (say, simple) expressions such as arithmetic expressions, it might appear inconsistent and confusing to the modeler, as operations with side-effects are allowed in some expressions only.

For these reasons, we decided to completely avoid such a redundant approach. In the limited scope of programming in the domain of models, we expect 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 at hand as a part of the imperative language and therefore a lot of programming can be done in a functional manner.

7. Related Work

As said in the introduction, our work does not make a contribution to the general field of programming languages. In general, the modular composition of programming languages and the embedding of imperative languages into higher-order functional languages has been researched for many years and the results are implemented in several languages.

The standard technique to implement an imperative language into a higher-order functional language are monads [33], a concept from category theory that has been widely applied, e.g., in Haskell, Scala, and Scheme. Monads are structures that represent computations, which can be chained. In functional programming languages, a monads consist of a type constructor, a *unit* function to encapsulate values into the monad, and a *bind* function to chain two monads to a larger computation. Haskell's *do* notation [39], for example, is just a syntactically convenient form for using monads. Monad transformers, another higher-order functional construct, allow to combine monads (to lift one monad into another). Concerning SOIL the language could be elegantly implemented in Haskell using the *State* and *Maybe* monads (the latter to represent errors). More generally, the modular composition of languages based on monads has been researched, e.g., in [26, 15]. As OCL does not support higher-order functions, this approach is not applicable (given that we do not want to change the language), although it could be studied, e.g., in HOL-OCL [4].

In similar vein, structural operational semantics (modular SOS) [34] address the composition of languages whose sematics is defined operationally [41, 47] (i.e., describing the individual computation steps). Whereas in SOS all rules may need to be reformulated whenever new constructs are added to the programming language (like adding side-effected expressions to OCL), in MSOS, only those rules need to be updated that are actually concerned by the new constructs. As for the monadic denotational approach, this approach is not applicable to OCL since there is no agreed deterministic operational semantics for OCL that clarifies, for example, the evaluation orders of expressions (which is left open for OCL). Furthermore the 'carrying around' of states through the evaluation process in actual OCL implementations would remain unsolved, too.

The specific problem of the embedding OCL into imperative modeling language has also been discussed in general lines by Siikarla et al. [32], who also argue for a separation of side-effected statements and side-effect free (pure) expression when embedding OCL into other languages.

8. Conclusion

In this paper we have addressed the reuse of OCL as a pure query language in an imperative programming language. We have discussed that extending OCL by expressions side-effects leads to problems on the semantic level (underspecification) and on the practical, implementation level (as existing implementations cannot be reused).

Our work focussed on working with the existing OCL standard and its implementations. Since OCL does not provide higher-order functions (like Haskell or Scala), imperative languages cannot be integrated in elegant ways (e.g., using Monads). We thus presented the language SOIL, which we have developed both driven by practical needs (for our tool USE) and as a means to study and demonstrate how one can integrate OCL safely into a programming language in the modeling context. We have shown the restrictions that are imposed by such a language design (e.g., when compared to ImperativeOCL), which, in our view, are acceptable in the context of model animation and model transformation.

In this sense, we hope that our work might contribute to the development of other OCL-based languages in the future, too.

References

[1] Abadi, M., Cardelli, L., 1996. A Theory of Objects. Springer-Verlag, New York.
[2] Akehurst, D., Patrascoiu, O., 2004. OCL 2.0 - Implementing the Standard for Multiple Metamodels. Electronic Notes in Theoretical Computer Science 102 (0), 21 – 41.
[3] Anastasakis, K., Bordbar, B., Georg, G., I.Ray, 2007. UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D. C., Weil, F. (Eds.), MoDELS 2007. Vol. 4735 of Lecture Notes in Computer Science. Springer, pp. 436–450.
[4] Brucker, A. D., Wolff, B., 2009. Semantics, calculi, and analysis for object-oriented specifications. Acta Inf. 46 (4), 255–284.
[5] Büttner, F., Gogolla, M., 2011. Modular Embedding of the Object Constraint Language into a Programming Language. In: da Silva Simão, A., Morgan, C. (Eds.), Formal Methods, Foundations and Applications - 14th Brazilian Symposium, SBMF 2011, São Paulo, Brazil, September 26-30, 2011, Revised Selected Papers. Vol. 7021 of Lecture Notes in Computer Science. Springer, pp. 124–139.

[6] Cabot, J., Clarisó, R., Riera, D., 2007. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ASE '07. ACM, New York, NY, USA, pp. 547–548.
URL `http://doi.acm.org/10.1145/1321631.1321737`

[7] Cabot, J., Teniente, E., 2007. Transformation Techniques for OCL Constraints. Science of Computer Programming 68 (3), 179–195.

[8] Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S., 2004. Ensuring UML Models Consistency Using the OCL Environment. Electronic Notes in Theorethical Computer Science 102, 99–110.

[9] Clark, T., 1999. Type Checking UML Static Diagrams. In: France, R. B., Rumpe, B. (Eds.), UML'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings. Vol. 1723 of Lecture Notes in Computer Science. Springer, pp. 503–517.

[10] Clavel, M., Egea, M., de Dios, M. A. G., 2008. Building an Efficient Component for OCL Evaluation. ECEASST 15.

[11] Clavel, M., Egea, M., de Dios, M. A. G., 2009. Checking Unsatisfiability for OCL Constraints. Electronic Communications of the EASST 24.

[12] Cuadrado, J. S., Jouault, F., Molina, J. G., Bézivin, J., 2008. Deriving OCL Optimization Patterns from Benchmarks. ECEASST 15.

[13] Gogolla, M., Büttner, F., Richters, M., 2007. USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34.

[14] Gogolla, M., Kuhlmann, M., Büttner, F., 2008. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (Eds.), Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings. Vol. 5301 of Lecture Notes in Computer Science. Springer, pp. 446–459.

[15] Hudak, P., 1998. Modular Domain Specific Languages and Tools. In: Proceedings of the Fifth International Conference on Software Reuse. IEEE Computer Society Press, pp. 134–142.

[16] Hußmann, H., Demuth, B., Finger, F., 2002. Modular Architecture for a Toolset supporting OCL. Science of Computer Programming 44 (1), 51–69.

[17] Jeanneret, C., Eyer, L., Markovic, S., Baar, T., 2006. RoclET – Refactoring OCL Expressions by Transformations. In: Software & Systems Engineering and their Applications,19th International Conference, ICSSEA 2006.

[18] Jiang, K., Zhang, L., Miyake, S., 2008. Using OCL in Executable UML. ECEASST 9.

[19] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., 2008. ATL: A model transformation tool. Science of Computer Programming 72 (1-2), 31–39.

[20] Jouault, F., Kurtev, I., 2005. Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. Online, `http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev__transforming_models_with_atl.pdf`.

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

[22] Kolovos, D. S., Paige, R. F., Polack, F., 2006. The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (Eds.), ECMDA-FA. Vol. 4066 of Lecture Notes in Computer Science. Springer, pp. 128–142.

[23] Krieger, M. P., Knapp, A., 2008. Executing Underspecified OCL Operation Contracts with a SAT Solver. ECEASST 15.

[24] Kyas, M., 2005. An extended type system for ocl supporting templates and transformations. In: Steffen, M., Zavattaro, G. (Eds.), Formal Methods for Open Object-Based Distributed Systems, 7th IFIP WG 6.1 International Conference, FMOODS 2005, Athens, Greece, June 15-17, 2005, Proceedings. Vol. 3535 of Lecture Notes in Computer Science. Springer, pp. 83–98.

[25] Kyas, M., Fecher, H., de Boer, F. S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H., 2005. Formalizing UML models and OCL constraints in PVS. Electronic Notes in Theoretical Computer Science 115, 39–47.

[26] Liang, S., Hudak, P., Jones, M., 1995. Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '95. ACM, New York, NY, USA, pp. 333–343.
URL `http://doi.acm.org/10.1145/199448.199528`

[27] Markovic, S., Baar, T., 2008. Refactoring OCL annotated UML class diagrams. Software and System Modeling 7 (1), 25–47.

[28] MDT, 2012. Eclipse, Model Development Tools (MDT), OCL. website, `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[29] Mellor, S. J., 2002. Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley.

[30] Mellor, S. J., Scott, K., Uhl, A., Weise, D., 2004. MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Boston.

[31] Meyer, B., 1992. Eiffel: The Language. Prentice-Hall.

[32] Mika Siikarla, J. P., Selonen, P., 2004. Combining OCL and Programming Languages for UML Model Processing. In: Schmitt, P. (Ed.), Proceedings of the Workshop, OCL 2.0 – Industry Standard or Scientific Playground. Vol. 102. Elsevier, pp. 175–194.

[33] Moggi, E., 1991. Notions of computation and monads. Inf. Comput. 93 (1), 55–92.

[34] Mosses, P. D., 2004. Modular structural operational semantics. J. Log. Algebr. Program. 60-61, 195–228.

[35] OMG, 2011. Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language, FTF - Beta 1 (Document ptc/2010-10-05). Object Management Group, Inc., Internet: `http://www.omg.org`.

[36] OMG, 2011. Meta Object Facility (MOF) 2.0 Query/Views/Transformation Specification, version 1.1 (Document formal/2011-01-01). Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`.

[37] OMG, 2011. OMG Unified Modeling Language (OMG UML), Superstructure (Document formal/2011-08-06). Object Management Group, Inc., Internet: `http://www.omg.org`.

[38] OMG, 2012. Object Constraint Language Specification, version 2.3.1 (Document formal/2012-01-01).

[39] O'Sullivan, B., Goerzen, J., Stewart, D., 2008. Real World Haskell: Code You Can Believe In. O'Reilly Media, Incorporated.

[40] Parr, T., 2007. The Definitive Antlr Reference: Building Domain-Specific Languages (Pragmatic Programmers). Pragmatic Bookshelf.

[41] Plotkin, G. D., 1981. A structural approach to operational semantics.

[42] Poppe, K.-M., 2011. Specification of an order management system with uml, ocl and soil [beschreibung eines auftragsverwaltungssystems mit uml, ocl und soil]. Master's thesis, University of Bremen, available online, `http://www.db.informatik.uni-bremen.de/publications/intern/poppe.pdf`.

[43] Richters, M., Gogolla, M., 2002. Ocl: Syntax, semantics, and tools. In: Clark, T., Warmer, J. (Eds.), Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Vol. 2263 of Lecture Notes in Computer Science. Springer, pp. 42–68.

[44] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2008. EMF: Eclipse Modeling Framework, 2nd Edition. Addison-Wesley Longman, Amsterdam.

[45] USE, 2012. The UML-based Specification Environment. homepage, `http://www.db.informatik.uni-bremen.de/`.

[46] Warmer, J. B., Kleppe, A. G., 2003. The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition. Addison-Wesley.

[47] Winskel, G., 1993. The formal semantics of programming languages: an introduction. MIT press.