

A New Type Checking Approach for OCL Version 2.0 ?

Andy Schürr

Institute for Software Technology
University of the Federal Armed Forces, Munich
D-85577 Neubiberg, Germany
Andy.Schuerr@unibw-muenchen.de,
URL: <http://ist.unibw-muenchen.de/schuerr>

Abstract. The Object Constraint Language OCL is an integral part of UML, the Unified Modeling Language standard. It has been added to Rational's UML core as a logic-based sublanguage for the definition of integrity constraints (invariants) on class diagrams as well as for the definition of pre- and postconditions of operations. Despite of the fact that OCL is called a statically typed language its type checking rules are not precisely defined in the UML standard version 1.3. Furthermore, they have certain deficiencies concerning the treatment of collection manipulating operations. This paper compares three different approaches for the definition of modified OCL type checking rules, selects one of these approaches as the most appropriate one and explains the new type checking rules for this approach in more detail. All presented proposals are based on our experiences with the design of a rather similar statically typed constraint language that is part of the graph transformation language PROGRES.

1 Introduction

The *object constraint language OCL* is an integral part of the Unified Modeling Language Standard UML [8] for the logic-based definition of class invariants or pre- and postconditions of operations [19]. In its current form OCL suffers from the same problems as many other parts of the UML standard: it possesses neither a precise static nor a precise dynamic semantics definition [14]. These are the reasons why groups of researchers are now active to refine and redesign parts of OCL in order to influence the definition of a new OCL version as part of the forthcoming UML 2.0 standard [5, 13].

It is the purpose of this paper to apply our experiences with the development of the *graph transformation language PROGRES* to improve OCL. PROGRES is a visual, executable specification language that combines a subset of UML class diagrams for the definition of object structures with graph transformation rules for the definition of object structure manipulations and OCL-like path expressions for the definition of integrity constraints and queries [18].

The PROGRES *path expression sublanguage* is similar to OCL w.r.t. the following properties:

- It combines arithmetic expressions, predicate logic formulas and so forth with path expressions for navigation along associations.
- It may be used for the definition of integrity constraints attached to a single class of objects and for the definition of pre- and postconditions of operations.
- It is therefore related to UML-like class diagrams and object manipulating operations in the same way as OCL.
- It distinguishes between partially defined and always defined expressions as well as between single object and collection returning path expressions, too.

On the other hand, there exists a long list of significant differences between PROGRES path expressions and OCL:

1. PROGRES path expressions have a well-defined set of type checking rules expressed as predicate logic formulas that avoid some type checking problems of OCL version 1.3.
2. Furthermore, PROGRES has a precisely defined dynamic semantics based on nonmonotonic reasoning and fixpoint theory.
3. Its dynamic semantics definition distinguishes between terminating computations that return the undefined result `nil` and nonterminating computations with unknown results.
4. Partially defined Boolean expressions, which caused the introduction of a three-valued logic in OCL, are disallowed; this is due to our experience that a three-valued logic often leads to rather unexpected results.
5. Functional abstraction of ordinary expressions and path expressions is supported by using different syntactic constructs.
6. Operators for the definition of default values (for partially defined subexpressions), building the transitive closure, conditional iteration etc. are available that are missing in OCL.
7. The OCL-like textual path expression sublanguage of PROGRES is complemented by a graphical sublanguage which is closely related to the structural part of UML collaboration diagrams.
8. Attributed associations, n-ary associations, bags and sequences are not supported, despite of the fact that PROGRES users often complain about the lack of these OCL features.

As a consequence, it was quite tempting to start a research project which refines the wide-spread OCL standard based on our experiences with the more or less unknown specification language PROGRES. First results concerning the addition of new operators to OCL and a visual sublanguage based on UML collaboration diagrams are presented in [16] and will not be repeated here. In the following we will focus our interest on the first topic mentioned above, i.e. the construction of appropriate *OCL type checking rules* based on our experiences in this area, the vast body of knowledge about type checking polymorphic specification and programming languages in general [2, 3], and a recently published paper about the OCL language's type system [4]. This paper presents a precise definition of the OCL type checking rules for the first time, but offers no solutions for the problems addressed in this contribution concerning the treatment of operations on collections (sets) of different types.

Our discussion of the addressed type checking problems is based on previously published results in a workshop proceedings [17]. For sake of completeness these results will be repeated here (in more detail). Furthermore, these results will be used in a modified form for the first time to define a set of type-checking rules for a representative subset of OCL.

The rest of this chapter is organized as follows: Section 2 explains the problems with the currently valid version of the OCL type checking rules for collection operators like `union`, `intersection`, and `includesAll`. Furthermore, it shows that variants of these problems affect the type checking rules for the comparison of objects or collections of objects, too.

Section 3 presents a straight-forward solution for the problems explained in Section 2 that follows the lines of the type checking approach invented for the predicate-logic-based language LOGIN [2]. This solution relies on the fact that class hierarchies have to be lattices, i.e. that any pair of classes possesses at most one smallest common superclass and at most one greatest common subclass (`OclAny` or `OclNil` in the worst case).

Section 4 presents a slightly different solution for the presented type checking problems that works with the powerset of all OCL types and avoids thereby the restriction of class hierarchies to lattices. Type checking based on sets of types is an already established approach, too. It is for instance used to define the type checking rules of the language TyCOON [12]. Both solutions of the type checking problem are closely related to each other; it is always possible to transform a given class hierarchy into a (for the programmer hidden) lattice that can be used for type checking purposes [1]. The additional classes of this lattice represent the needed nonsingleton sets of the type checking approach presented in Section 4.

Unfortunately, both approaches are too complex for the average OCL user who does not want to care about class lattices or sets of types. Therefore, Section 5 presents yet another solution that (1) works for all kinds of class hierarchies, (2) returns the same results as the type checking rules of the OCL standard version 1.3 as long as the standard rules do not reject a given OCL expression (and vice versa), and (3) simply determines a more general supertype where the type checking rules of Section 4 process nonsingleton sets of types. It is our opinion that this type checking approach constitutes the proper compromise between the restrictiveness and simplicity of the type checking rules of the OCL standard and the expressiveness and complexity of the solutions presented in Section 3 and 4 of this contribution.

Section 6 demonstrates for a core subset of OCL how the type checking approach introduced in Section 5 may be used to define the type checking rules for OCL version 2.0. Based on the examples presented here it is rather straightforward to define the missing type checking rules for the rest of OCL, too.

Finally, Section 7 summarizes the presented three new OCL type checking approaches and discusses planned future work activities.

2 Problems with Type Checking OCL Expressions

We believe that the implicitly defined type checking rules for OCL version 1.3 in [8] often return unwanted results and should be changed in the future version 2.0. Let us assume that the OCL types (classes) `Employee` (of our University) and `Student` are subtypes of `Person` as shown in Fig. 1. Furthermore, let us assume that an OCL expression $\langle T \rangle \text{Expr}$ computes an object which is a member (direct or indirect instance) of the type $\langle T \rangle$, and that an OCL expression $\langle T \rangle \text{SExp}$ computes a set of members of the type $\langle T \rangle$. Using these writing conventions we know that `EmployeeExpr` returns a member of type `Employee` (an instance of type `Employee` itself or an instance of `Professor` or `Assistant`), whereas `StudentSExp` returns a set of members of type `Student`.

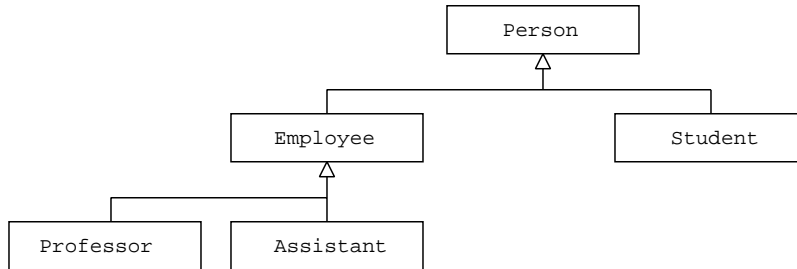


Fig. 1. A simple version of the running example.

Furthermore, we assume that

- the functor `Eval[Expr]` *evaluates* the given OCL expression and returns either an undefined result `Nil` or a single element or a collection of elements if provided with an additional environment parameter for the regarded object structure, used parameter values, etc.,
- the functor `Type[Expr]` determines a unique *static type* of the regarded expression or issues a type checking error by returning an undefined type which we call `OclNil`,
- the actual type of any element computed by `Eval[Expr]` always *conforms to* the static type returned by `Type[Expr]`,
- the set of all expressions with type `OclNil` is a (decidable) subset of the set of expressions with an undefined value `Nil`,
- and that `Type[Expr]` is the most specific type w.r.t. the “conforms to” partial order known at compile time that fulfills the other requirements mentioned above.

The static type of an OCL expression may be modeled as a tuple of the following form:

$$\text{Type}[\text{Expr}] = (\text{Type}[\text{Expr}].\text{Sort}, \text{Type}[\text{Expr}].\text{Name}) \in \text{Sort} \times \text{Name}$$

where

$Sort = \{\text{simple}, \text{partial}, \text{collection}, \text{bag}, \text{set}, \text{sequence}\}$
 $Name = BasicTypes \cup Classifiers \cup \{0clNil\}$
 $BasicTypes = \{\text{Real}, \text{Integer}, \text{String}, \text{Boolean}\}$
 $Classifiers =$ the set of all classifiers of the related class diagram(s)

This representation of OCL types is not totally consistent with the OCL standard because of the addition of a “smallest” type `0clNil` that represents always undefined or illegal (wrong typed) expressions and the introduction of `partial` expressions. The standard is rather vague concerning the treatment of expressions which either return a single well-defined element or an undefined result as e.g. the navigation to an association end with multiplicity `0..1`. It is unclear whether such a *partial expression* has to be treated (1) like an expression that returns a single instance which may be `nil`, or (2) like an expression that returns a collection of results which contains no or one element.

All remaining details of the static type representation introduced above are consistent with the OCL standard. We distinguish between `simple` expressions that have a single well-defined element as value and `collection` expressions that return a collection of values. The value of a simple expression may either be an element of one of the four basic types `Real, ...` or an object (instance) of a defined UML classifier. Collections of values are either ordered sequences or unordered sets or bags that may contain multiple occurrences of the same element. The type `0clAny` is used as the supertype of all possible types of simple expressions.

Based on these assumptions the following type hierarchy definition (type conformance) rules are introduced (cf. [19]):

- `Type1` conforms to `Type2`
if `Type1.Sort` is compatible with `Type2.Sort` and `Type1.Name` is a subtype of `Type2.Name`.
- `Sort1` is compatible with `Sort2` if `Sort1 = Sort2`.
- `Sort1` is compatible with `Sort2`
if `Sort2 = collection` and `Sort1 ∈ {bag, set, sequence}`.
- `Sort1` is compatible with `Sort2`
if `Sort1 = partial` and `Sort2 = sequence`¹.
- `Name1` is a subtype of `Name2` if `Name1 = Name2`.
- Any type `Name ∈ BasicTypes ∪ Classifiers` is a subtype of `0clAny`.
- `0clNil` is a subtype of any type `Name ∈ BasicTypes ∪ Classifiers`.
- `Integer` is a subtype of `Real`.

¹ A partial expression is not compatible with a simple expression or a set expression or a bag expression for the following reasons: the OCL standard states that navigations along associations with multiplicity unequal to `1..1` always return a sequence of elements. However, it is tempting to require that partial expressions are compatible with any sort of collections and to permit simple expressions where partial expressions are allowed. But these rules would have the side effect that simple expressions may be used where collections are expected, i.e. destroy OCL's philosophy to distinguish between operations on simple values and operations on collections.

- Name1 is a subtype of Name2
if Name1 and Name2 are UML classifiers and Name2 generalizes Name1.
- The “conforms to” relationship is a partial order.

The suggested treatment of OCL types disregards type and expression parameters of OCL operators for the following reasons: OCL offers some operators that require types as input or return types as their results. However, it is still unclear whether OCL has the intention to treat *types as real first-order objects* and whether its future versions will offer *all* needed operators for accessing all (meta layer) properties of OCL types (UML classifiers). Today available operators return the attributes, association ends, and operations for a given type as collections of strings. No means are available for retrieving the properties of these strings, i.e. for determining the type of an attribute or the parameter profile of an operation or the multiplicity of an association end. It makes therefore no sense to present a type system that distinguishes between first-order types of object instances and second-order types of type instances as long as these important details are not clarified.

Concerning the treatment of *expression parameters* of OCL operators the situation is as follows: An operator like `iterate` that applies a given expression parameter to a collection of elements simply requires that this expression parameter has the type `OCLExpression`. The fact that the expression parameter references variables defined by the surrounding `iterate` expression is not taken into account by the OCL type system. Therefore, we will not make any attempts to define parameter profiles for OCL operators like `iterate` themselves, but restrict our interest to type checking rules for expressions containing applied/expanded occurrences of these operators.

Based on these assumptions we are now able to explain some of the problems with the currently used type checking rules of OCL. Let us consider some set manipulating expressions such as

`XExpr->union(YExpr)`

which computes the union of a set of X members and a set of Y members and

`XExpr->intersection(YExpr)`

which computes the intersection of a set of X members and a set of Y members.

The signatures (parameter profiles) of the involved set operations have about the following form:

`Set(T)->union(set2:Set(T)) : Set(T)`

`Set(T)->intersection(set2:Set(T)) : Set(T)`

These signatures require for both set manipulating expressions that the actual type (name) X is bound to the formal parameter type T and that the type Y of the second parameter is therefore a subtype of X. The result type of both expressions is `Set(X)`, i.e.

`Type[XExpr->union(YExpr)] =`

`Type[XExpr->intersection(YExpr)] = (set,X)`

This has the consequence that the OCL expressions

`EmployeeSEExpr->union(PersonSEExpr)` and
`EmployeeSEExpr->intersection(PersonSEExpr)`

are illegal (`Person` is not a subtype of `Employee`), whereas the expressions

`PersonSEExpr->union(EmployeeSEExpr)` and
`PersonSEExpr->intersection(EmployeeSEExpr)`

are legal and possess the static type `(set, Person)`. These results are in contradiction to our intuition that

- the mathematical union or intersection of $S1$ and $S2$ should be equivalent to the union or intersection of $S2$ and $S1$,
- the union of a set of `Employee` members with a set of `Person` members is computable and returns a set of `Person` members in any case,
- the intersection of a set of `Employee` members and a set of `Person` members always returns a set of `Employee` members (and never a `Person` member which is not an `Employee` member).

Similar problems occur when we study the definition of the signatures of the operations `includes`, `includesAll`, and the identity test for two objects under the assumption that objects are direct instances of a single class²:

- `EmployeeSEExpr->includes(StudentSEExpr)` tests whether the result computed by `StudentSEExpr` is an element of the set computed by `EmployeeSEExpr`. Regarding the class hierarchy of Fig. 1 we know that this expression is always `false` due to the fact that `Employee` and `Student` do not possess a common subclass. Nevertheless this expression is legal w.r.t. the signature `collection(T)->includes(object:OclAny) : Boolean` of the `includes` operation.
- `EmployeeSEExpr->includesAll(StudentSEExpr)` on the other hand is an illegal operation even if `Employee` and `Student` would possess a common subclass. This is due to the fact that `includesAll` possesses the signature `collection(T)->includesAll(c2: collection(T)) : Boolean`.
- For similar reasons `EmployeeSEExpr = StudentSEExpr` is a legal expression which always returns `false` w.r.t. the class hierarchy of Fig. 1,
- whereas `EmployeeSEExpr = StudentSEExpr` is illegal as long as `Student` is not a subtype of `Employee`.

As a consequence, the standard type checking rules *are not complete* w.r.t. the OCL language's dynamic semantics as required in [4]. Some reasonable OCL expressions with a well-defined dynamic semantics are illegal w.r.t. the standard type checking rules. Even worse, the given rules often return types which are too

² UML permits objects that are direct instances of more than one class. However, most if not all UML tools and UML users rely on the fact that objects are instances of a single class which does not change during an object's life time. Therefore, OCL should at least possess a more sophisticated type checking mode which makes use of this widespread assumption and disallows e.g. the comparison of expressions with unrelated static types.

general and they return different results for expressions which obviously have the same dynamic semantics.

Therefore, some OCL explanations already use a different interpretation for the signatures of OCL collection operations such as `union` and `intersection`. They require for expressions like

```
XSEpr->someOperation(YSEpr)
with someOperation ∈ {union, intersection, ...}
```

the existence of a (smallest) common supertype of X and Y which is used as parameter T of the involved signature (template)

```
set(T)->someOperation(set2 : Set(T)): Set(T).
```

Such a smallest common supertype always exists due to the fact that all non-collection types are subtypes of `UclAny`. At a first glance this interpretation of the OCL standard solves almost all problems mentioned above—as long as the multiple inheritance concept is not used.

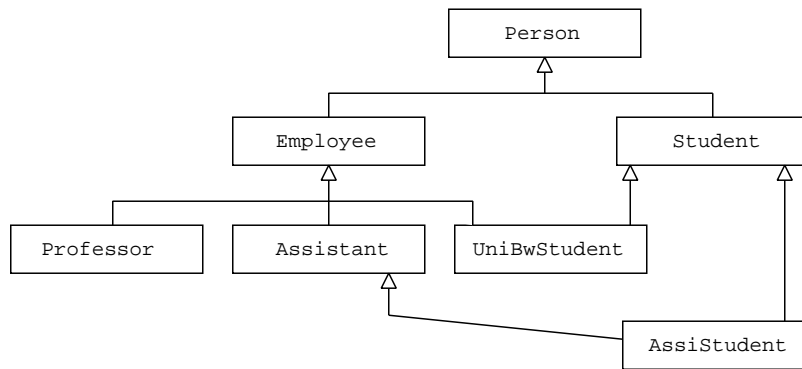


Fig. 2. A class hierarchy with multiple inheritance.

Fig. 2 presents one example of a UML class diagram that uses multiple inheritance. Both `Assi(stant)Student` and `UniBwStudent` are subtypes (subclasses) of `Employee` and `Student`. Probably such a class diagram is ill-designed³ but it is legal. Regarding this class diagram we can explain our problems with the suggested new OCL type checking rules. Let us start with computing the types of the two expressions

```
AssiStudentSEpr->union(UniBwStudentSEpr)
EmployeeSEpr->intersection(StudentSEpr)
```

Using the new “smallest common supertype” rule we have to conclude that the union of the `AssiStudentSEpr` and the `UniBwStudentSEpr` has not a sin-

³ For further details concerning a proper treatment of temporary object roles such as being a student or a professor the reader is referred to [10].

gle but two smallest common supertypes `(set,Employee)` and `(set,Student)` whereas the intersection of the `EmployeeSEExpr` and the `StudentSEExpr` has the most general type `(set,Person)`. Both results are rather unwanted. In the case of the union expression we either have to work with a set of OCL types or we have to replace the type set `{(set,Employee), (set,Student)}` by the common supertype `(set,Person)`. In the case of the intersection we have lost the information that all elements of the resulting set of `Person` elements are members of the type `Employee` and the type `Student`.

Similar problems occur, when we regard the (symmetric) difference of two collections or the inclusion or exclusion of single elements from collections. In many cases the standard OCL type checking rules as well as the suggested new type checking rules force the OCL user to add type casts at various places. Furthermore, the proposed type checking rules do not only return useless type information in many cases but still permit the construction of many useless expressions that could be recognized at compile time.

3 Type Checking with Type Lattices

It is possible to circumvent some of the type checking problems discussed in Section 2 by requiring that any two types have at most one smallest common supertype `scs` and at most one greatest common subtype `gcs`. These restrictions together with the existence of a greatest type (name) `oclAny` and a smallest type (name) `oclNil` ensure that all constructed type hierarchies are lattices (in the mathematical sense of the word).

Therefore, the class diagram of Fig. 2 is illegal, which maybe recognized by an incremental lattice checking algorithm as implemented in the PROGRES programming environment [18]. An automatically working completion algorithm adopted from [1] may be used to transform the class diagram of Fig. 2 into the class diagram of Fig. 3 by adding a single class `Emp1Student` and by redirecting some generalization relationships.

Relying on the automatically constructed class diagram of Fig. 3 we may introduce OCL type checking rules based on the following definitions of two binary functions `scs` and `gcs` on types:

- `scs(t, t')` computes *the unique* smallest common supertype of `t` and `t'`; it returns `oclAny` in the worst case.
- `gcs(t, t')` computes *the unique* greatest common subtype of `t` and `t'`; it returns `oclNil` in the worst case.

The following OCL type checking rules are still asymmetric w.r.t. the treatment of the *sort* of the first and the second argument of collection operations like `union` or `intersection` of sets in order to avoid any unnecessary inconsistencies with the currently valid OCL type checking rules. These rules require that the `union` of a `set` and a `bag` returns a `set` whereas the `union` of a `bag` and a `set` returns a `set`. It is worthwhile to discuss a modification of these rules such that the expression `sort simple` is compatible with `partial` and the expression `sort`

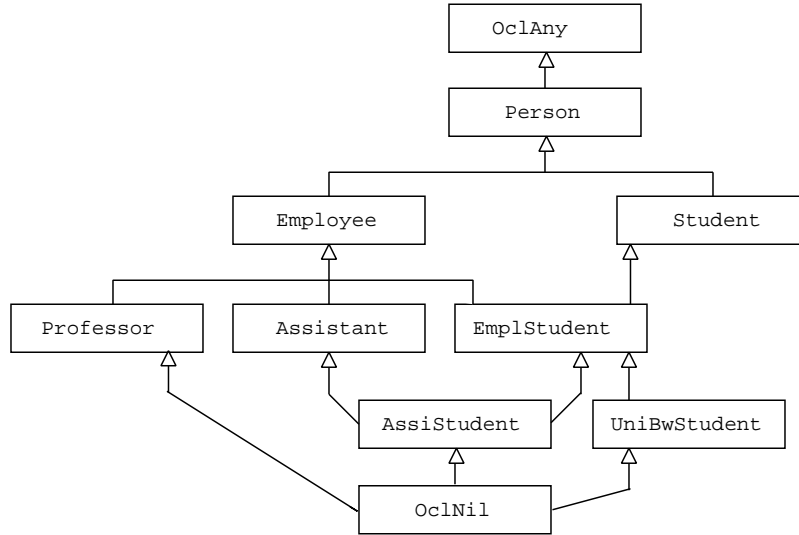


Fig. 3. Redesigned class hierarchy with lattice property.

set is compatible with **bag**. Based on these extensions the union of a **set** with a **bag** and a **bag** with a **set** would return a **bag**. Furthermore, the union of a **set** or **bag** expression with a **simple** or **partial** expression would be permitted, too; its result would be a **set** or **bag**, respectively.

Avoiding the just discussed extensions and sticking to the standard as close as possible we get the following type checking rules for **union**, **intersection**, and the identity operator = defined on simple elements and collections of elements:

1. $\text{Type}[\text{expr1} \rightarrow \text{union}(\text{expr2})] :=$
 $(\text{set}, \text{scs}(\text{Type}[\text{expr1}].\text{Name}, \text{Type}[\text{expr2}].\text{Name})),$
 if $\text{Type}[\text{expr1}].\text{Sort} = \text{set}$ and $\text{Type}[\text{expr2}].\text{Sort} \in \{\text{set}, \text{bag}\}.$
2. $\text{Type}[\text{expr1} \rightarrow \text{union}(\text{expr2})] :=$
 $(\text{bag}, \text{scs}(\text{Type}[\text{expr1}].\text{Name}, \text{Type}[\text{expr2}].\text{Name})),$
 if $\text{Type}[\text{expr1}].\text{Sort} = \text{bag}$ and $\text{Type}[\text{expr2}].\text{Sort} \in \{\text{set}, \text{bag}\}.$
3. $\text{Type}[\text{expr1} \rightarrow \text{intersection}(\text{expr2})] :=$
 $(\text{set}, \text{gcs}(\text{Type}[\text{expr1}].\text{Name}, \text{Type}[\text{expr2}].\text{Name})),$
 if $\text{Type}[\text{expr1}].\text{Sort} = \text{set}$ and $\text{Type}[\text{expr2}].\text{Sort} \in \{\text{set}, \text{bag}\}.$
4. $\text{Type}[\text{expr1} \rightarrow \text{intersection}(\text{expr2})] :=$
 $(\text{bag}, \text{gcs}(\text{Type}[\text{expr1}].\text{Name}, \text{Type}[\text{expr2}].\text{Name})),$
 if $\text{Type}[\text{expr1}].\text{Sort} = \text{bag}$ and $\text{Type}[\text{expr2}].\text{Sort} \in \{\text{set}, \text{bag}\}.$
5. $\text{Type}[\text{expr1} = \text{expr2}] := (\text{simple}, \text{Boolean}),$
 if $\text{gcs}(\text{Type}[\text{expr1}].\text{Name}, \text{Type}[\text{expr2}].\text{Name}) \neq \text{OclNil}$
 and $\text{Type}[\text{expr1}].\text{Sort} = \text{Type}[\text{expr2}].\text{Sort} \neq \text{collection}.$
6. Any expression without a matching rule is illegal. The same is true for expressions to which the type (S, OclNil) is assigned where S may be **simple** or **partial** or ...

Applying these rules to the class diagram of Fig. 3 and some expressions introduced in Section 2 we get the following results:

```
Type[AssiStudentExpr->union(UniBwStudentExpr)] =
  (set,scs(AssiStudent,UniBwStudent)) = (set,EmplStudent)
```

and

```
Type[EmployeeSEExpr->intersection(StudentSEExpr)] =
  (set,gcs(Employee,Student)) = (set,EmplStudent)
```

Furthermore, the presented rules now reject expressions like

```
ProfessorExpr = AssistentExpr
```

due to the fact that `Professor` and `Assistent` have no (greatest) common subtype except of `oclNil`.

The main drawbacks of the presented solution are the requirements that type hierarchies have to be lattices and that objects are direct instances of a single type (class). These requirements often enforce the construction of better designed class hierarchies. However, our experiences show that it is difficult to convince OO modelers that type hierarchies should be lattices and to teach them how to transform a given type hierarchy into a lattice. Even if they are supported by tools which perform this task, they have difficulties to understand the output of such a transformation process, i.e. the needs for adding an exponential number of additional types in the worst case. Therefore, we present in the following section yet another variant of OCL type checking rules. These rules manipulate sets of OCL types instead of relying on the lattice property of class hierarchies.

4 Type Checking with Type Sets

One solution for avoiding the lattice restriction for type hierarchies is quite obvious: a hidden preprocessing phase checks the lattice property for the given UML class diagrams and adds the still needed intermediate classes without revealing their existence to the end user as already suggested in [1]. Whenever the type checking algorithm computes one of these intermediate types for a given OCL expression it displays its set of smallest user defined supertypes. From a theoretical point of view the lattice construction phase may even be omitted and the type checking rules may directly manipulate *sets of OCL types*. For this purpose the general “type checking axioms” of Section 2, which guide the design of our type checking rules, have to be modified as follows:

- the functor `Eval[Expr]` evaluates the given OCL expression and returns either an undefined result or a single element or a collection of elements (if provided with an additional environment parameter not considered here),
- the functor `Type[Expr]` determines a *set of* static types of the regarded expression or issues a type checking error by returning an *empty set of types*.
- the actual type of any element computed by `Eval[Expr]` conforms to *all static types* returned by `Type[Expr]`,

- the type checking rules used for the definition of the functor `Type` assign *the empty set of types* to expressions with an always undefined value,
- and the type checking rules compute *the smallest set* of the most specific types w.r.t. the “conforms to” partial order that fulfills the requirements mentioned above.

Based on these assumptions the type checking rules for the operators `union` and `intersection`—restricted to arguments of sort `set` for simplicity reasons—have the following form:

```
Type[expr1->union(expr2)] := (set, rmSuper(TS))
```

where

```
TS = (allSuper(Type[expr1].Name) ∩ allSuper(Type[expr2].Name))
```

Furthermore,

```
Type[expr1->intersection(expr2)] := (set, rmSuper(T))
```

where

```
TS = allSuper(Type[expr1].Name) ∪ allSuper(Type[expr2].Name)
```

The used auxiliary functions `rmSuper` and `allSuper` are defined as follows using OCL and the functions `directSupertypes` and `directSubtypes` of the UML meta model:

```
allSuper(set) = set->union(directSupertypes.allSuper)
-- allSuper computes the set of all direct and indirect
-- supertypes of a set of types including the given set of
-- types itself but without the most general type OclAny

rmSuper(set) =
set->reject(directSubtypes->intersection(set)->notEmpty)
-- rmSuper removes all types from set with direct subtypes
-- in the set
```

Returning to our running example and the class diagram of Fig. 2 these new type checking rules work as follows:

```
Type[AssiStudentSEExpr->union(UniBwStudentSEExpr)] =
(set, rmSuper(TS)) = (set, {Employee, Student})
```

where

```
TS = allSuper(AssiStudent) ∩ allSuper(UniBwStudent) =
= {AssiStudent, Assistant, Employee, Student, Person} ∩
{UniBwStudent, Employee, Student, Person}
```

Furthermore,

```
Type[EmployeeSEExpr->intersection(StudentSEExpr)] =
(set, rmSuper(TS)) = (set, {Employee, Student})
```

where

```
TS = allSuper(Employee) ∪ allSuper(Student)
= {Employee, Person} ∪ {Student, Person}
```

At a first glance the new type checking rules seem to be wrong for the following reasons: the type set of a `union` expression is computed by building the *intersection* of the types of its subexpressions whereas the type set of a `intersection` expression is computed by building the *union* of the types of its subexpressions. However, we have to keep in mind that an element of the union of two sets is a member of all the static types of the first argument *or* a member of all the static types of the second argument but not an element of the union of all static types of both arguments in the general case. Therefore, we have to replace the set of types of the first and the second argument by the smallest set of *common* supertypes. For this purpose we (1) determine the supertypes of the types of the arguments, (2) build the intersection, and (3) remove all redundant supertypes of the resulting set of types.

Similarly, we know that any element in the intersection of two sets is a member of all static types of the first set *and* that it is simultaneously a member of all static types of the second set. Therefore, it is save to compute the type set of an intersection by (1) determining the supertypes of the involved arguments, (2) building the union of the just constructed type sets, and (3) removing all redundant supertypes of the resulting set of types.

The main advantage of using type sets in type checking rules—instead of building first a hidden lattice—is related to the reason why the `oclType` operator of previous OCL versions has been removed in version 1.3. It has been removed because UML allows models where one object is a direct instance of more than one type (classifier). As a consequence, the `oclType` operator might return a set of types in the general case instead of always returning a single type as one might expect. The set-oriented type checking approach has no longer any problems with objects which are direct instances of more than one type. As a consequence it is possible to reintroduce the `oclType` operator together with other means for reflection.

The fact that objects may be direct instances of more than one type (classifier) had an important impact on the definition of the above introduced type checking rule. These rules do not make any attempt to compute sets of greatest common subtypes of the types of the regarded subexpressions. Therefore, the computed type set of the intersection expression

```
Type[EmployeeSEExpr->intersection(StudentSEExpr)] =
  (set, {Employee, Student}) ≠ (set, {EmplStudent})
```

is independent of the fact whether the class diagram of Fig. 2 or the class diagram of Fig. 3 is regarded. An object may be a member (direct instance) of both types (classes) `Employee` and `Student` without being a member of a subtype (subclass) of these two types (classes). Therefore, the object set of all members (extension) of `EmplStudent` is a proper subset of the intersection of the object sets (extensions) of all members of `Employee` and all members of `Student` in the general case.

5 Type Checking with Approximated Type Sets

The previously presented type checking solutions are either too restrictive or too complex to be useful for the average OCL user. The solution based on lattices enforces UML users to redesign their class hierarchies in certain cases, the solution based on sets of types is very unnatural for software developers familiar with the type checking rules of programming languages. Therefore, we have to find another solution with about the following properties:

- The new type checking rules accept more expressions with a well-defined dynamic semantics than the standard rules.
- If the standard OCL type checking rules and the rules of this section both accept an expression then the type checking rules defined here compute either the same type or a subtype of the “old” type.
- The new rules are identical to the rules presented in Section 3 as long as the regarded type hierarchies are lattices (after completion with `oclAny` and `oclNil`).
- The new rules approximate nonsingleton type sets computed by the rules of Section 4 by a unique smallest common supertype that is `oclAny` in the worst case.

Type checking rules that fulfill these properties may be defined as follows for the operators `union` and `intersection` (restricted to sets):

```
Type[expr1->union(expr2)] :=
  (set, scs*(Type[expr1].Name, Type[expr2].Name))
Type[expr1->intersection(expr2)] :=
  (set, gcs*(Type[expr1].Name, Type[expr2].Name))
```

where

```
scs*(T1, T2) := approximate(set of smallest common supertypes of T1, T2)
gcs*(T1, T2) := approximate(set of greatest common subtypes of T1, T2)
approximate({T}) := T
approximate({T1, T2} ∪ TSet) := approximate(scs*(T1, T2) ∪ TSet)
```

Having presented these rules we have to prove that their definitions are sensible. First of all we have to show that the evaluation of the just introduced recursively defined functions `scs*` and `approximate` terminates for all possible inputs. This is true for the following reasons: The operator `scs*` either determines a set of *proper* supertypes of its two type arguments or it returns one of the two type arguments (if one argument is a supertype of the other one). Based on the assumption that type hierarchies are finite and that the supertype (subtype) relationship is a partial order we know that the approximation process will finally terminate with returning `oclAny` in the worst case.

Furthermore, it is easy to show that the new rules fulfill the properties required above:

- The new rules accept all expressions accepted by the rules of Section 3, which were already more liberal than the type checking rules of the standard in some cases.

- The standard OCL type checking rules require that the type of the second argument of a `union` or `intersection` conforms to the type of its first argument. Therefore, the type of the first argument of a legal `union` expression is always the smallest common supertype of the types of both arguments whereas the type of the second argument of a legal `intersection` expression is always the greatest common subtype of the types of both arguments. As a consequence the new rules compute for the `union` the same type and for the `intersection` a subtype of the type computed by the standard rules.
- As long as the regarded UML class hierarchy is a lattice no approximation of type sets by the operator `approximate` is needed. Under these assumptions the rules defined here compute the same type as the rules of Section 3.

Applying the new type checking rules to some OCL expressions of Section 3 we expect therefore the following results for the class hierarchy of Fig. 3:

```
Type[AssiStudentExpr->union(UniBwStudentExpr)] =
  (set, EmplStudent)
Type[EmployeeExpr->intersection(StudentExpr)] =
  (set, EmplStudent)
```

Applying the new rules to the same expressions regarding the class hierarchy of Fig. 2 yields the following results:

```
Type[AssiStudentExpr->union(UniBwStudentExpr)] =
  (set, approximate({Employee, Student} = (set, Person)
Type[EmployeeExpr->intersection(StudentExpr)] =
  (set, approximate({AssiStudent, UniBwStudent} = (set, Person)
```

The approximations of sets of smallest common supertypes or greatest common subtypes are sometimes unprecise. The type checking rules of this section require in some cases “downcasts” where the type checking rules of Section 4 would accept an OCL expression as type safe. Using the type checking rules of Section 4 any operation defined for all members of type `Employee` or defined for all members of type `Student` may be applied to the union expression regarded above. On the contrary, the new type checking rules introduced in this section rule out the application of operations that are not defined for all members of the type `Person`.

A similar problem occurs when we regard the `intersection` defined above. We know that all elements of the result set are either members of `AssiStudent` or `UniBwStudent`. To be on the safe side we may therefore apply operations (features) to the computed set of elements which are defined for both types `AssiStudent` and `UniBwStudent`. These are the inherited features of the set of their smallest common superclasses, `Employee` and `Student`. Again the type checking rules of Section 4 would permit the application of all operations defined for `Employee` or `Student`, whereas the type checking rules of this section permit the application of operations defined for the type `Person` only. This is the price we have to pay for manipulating single type names instead of sets of type names if the regarded type hierarchies are not lattices. However, we believe that OCL users will not have any problems with the type checking rules of this section for the following reasons:

- The new type checking rules require a considerable smaller number of downcasts than the standard rules.
- OO programmers are used to solve type checking problems by inserting type casts if needed.
- Software developers are told to avoid multiple inheritance whenever possible, i.e. developed type hierarchies are very often trees, which are lattices after the addition of the smallest type `oclNil`.

6 More Type Checking Rules for OCL

Based on the type checking approach presented in the previous section we will now propose a complete set of *type checking rules for a core subset of OCL*. Again, we are avoiding mathematical symbols as far as possible⁴ in order to make these rules readable for the average OCL user. However, it is a rather straightforward task to translate the rules given here into more formal notations which are usually used by experts for the definition of type theories.

Furthermore, we are not introducing an environment parameter which provides all needed details about regarded UML class diagrams and which binds variables of OCL expressions to their types. On the contrary, we are simply assuming that

- `Type[navigation]` := (S, T) , where T is the name of the involved classifier of the regarded binary association end and where
 - $S = \text{partial}$ if the multiplicity of the regarded association end is $0..1$,
 - $S = \text{simple}$ if the multiplicity is $1..1$, and
 - $S = \text{sequence}$, otherwise.
- `Domain[navigation]` := T , where T is the name of the classifier of the opposite end of the regarded binary association end.
- `Type[var]` := (S, T)
 - if the variable `var` is declared as `var: T` with $S = \text{simple}$
 - or declared as `var: S(T)` with $K \in \{\text{set}, \text{bag}, \text{sequence}, \text{collection}\}$.

Based on these assumptions we are now able to introduce the rules for navigating along associations (operations of classifiers and attributes may be treated similarly), comparing elements or collections of elements, and building the `union` and `intersection` of collections of elements. Furthermore we will present the type checking rules for the rather complex `iteration` operation, for conditional expressions, and for the various forms of type casts in OCL. These constructs form a kind of OCL core. They hopefully give the reader a sufficiently precise impression of how the type checking rules for the rest of OCL look like.

⁴ Steve Cook says in his foreword of the OCL book written by Jos Warmer and Anneke Kleppe that it was one of the most important design goals of OCL to avoid mathematical symbols in order to make it useful for a broad spectrum of software developers.

1. $\text{Type}[e.\text{navigation}] := (S, T)$, where
 - $T = \text{Type}[\text{navigation}].\text{Name}$
 - $S = \text{simple}$ if $\text{Type}[\text{navigation}].\text{Sort} = \text{Type}[e].\text{Sort} = \text{simple}$
 - $S = \text{sequence}$ if $\text{Type}[e].\text{Sort} \notin \{\text{simple}, \text{partial}\}$
 - $S = \text{sequence}$ if $\text{Type}[\text{navigation}].\text{Sort} \notin \{\text{simple}, \text{partial}\}$
 - $S = \text{partial}$ otherwise.

if $\text{Domain}[\text{navigation}]$ is a subtype of $\text{Type}[e].\text{Name}$.

A navigational expression returns a single well-defined element only if it is applied to a simple expression and if the regarded association end has multiplicity 1..1. It returns a sequence of objects if either the expression returns already a collection of elements or if the regarded association end has neither the multiplicity 0..1 nor the multiplicity 1..1. It is a partial expression which returns either a well-defined element or a “defined undefined” value otherwise.

2. $\text{Type}[e1 = e2] := (\text{simple}, \text{Boolean})$, if
 - $\text{gcs}*(\text{Type}[e1].\text{Name}, \text{Type}[e2].\text{Name}) \neq \text{oclNil}$ and if
 - $\text{Type}[e1].\text{Sort} = \text{partial}$ or if
 - $\text{Type}[e2].\text{Sort} = \text{partial}$ or if
 - $\text{Type}[e1].\text{Sort} = \text{Type}[e2].\text{Sort} \neq \text{collection}$

It is a matter of debate whether the comparison of two partial expressions or the comparison of a partial expression with an always defined expression should be permitted. The OCL standard probably treats partial expressions like sequences. It disallows therefore the comparison of a partial expression with an always defined expression but permits the comparison of a partial expression with a sequence. From our point of view the most liberal solution allows the comparison of partial expressions with any sort of expressions by treating undefined values like empty bags, sequences, or sets if required.

It is clear that the comparison of two expressions makes no sense if their types have no common subtype except of oclNil . In this case static analysis does already tell us that the comparison always delivers the result **false**. Furthermore, it is clear that the comparison of a simple expression with a collection or a set with a bag, etc. should be forbidden, thereby forcing the OCL user to insert conversions from sets to bags, ... explicitly.

3. $\text{Type}[e1 \rightarrow \text{union}(e2)] := (S, T)$, where
 - $T = \text{scs}*(\text{Type}[e1].\text{Name}, \text{Type}[e2].\text{Name})$
 - $S = \text{Type}[e1].\text{Sort} = \text{set}$ if $\text{Type}[e2].\text{Sort} \in \{\text{set}, \text{bag}\}$
 - $S = \text{Type}[e1].\text{Sort} = \text{bag}$ if $\text{Type}[e2].\text{Sort} \in \{\text{set}, \text{bag}\}$

The type checking rule for the union of sets or bags should be clear based on the explanations of the previous sections. Please note that OCL implicitly adds conversion functions where needed. If the first argument of the union of two expressions has the sort **set** and the second expression has the sort **bag** then the second argument is implicitly converted into a **set**. The same is true if the words **set** and **bag** are exchanged in the previous sentence.

4. $\text{Type}[e1 \rightarrow \text{intersection}(e2)] := (S, T)$, where
 $T = \text{gcs}*(\text{Type}[e1].\text{Name}, \text{Type}[e2].\text{Name})$
 $S = \text{Type}[e1].\text{Sort} = \text{set}$ if $\text{Type}[e2].\text{Sort} \in \{\text{set}, \text{bag}\}$
 $S = \text{Type}[e1].\text{Sort} = \text{bag}$ if $\text{Type}[e2].\text{Sort} \in \{\text{set}, \text{bag}\}$
 The type checking rule for the `intersection` of sets or bags should be clear based on the explanations of the previous sections.
5. $\text{Type}[\text{coll} \rightarrow \text{iterate}(e: T1; r: T2 = e1 \mid e2)] := ST$, where
 $ST = \text{Type}[r]$ if
 $\text{Type}[\text{coll}].\text{Name}$ is subtype of $\text{Type}[e].\text{Name}$ and
 $\text{Type}[\text{coll}].\text{Sort} \neq \text{simple}$ and $\text{Type}[e].\text{Sort} = \text{simple}$ and
 $\text{Type}[e1]$ conforms to $\text{Type}[r]$ and $\text{Type}[e2]$ conforms to $\text{Type}[r]$.
 The type checking rule for the `iteration` operator is rather straightforward if we regard its dynamic semantics definition. The operator assigns the value of the expression `exp1` to the accumulator variable `r` of type `T2`. Furthermore, it assigns one element of the given collection `coll` after the other to the loop variable `e` of type `T1`. Based on the just computed value of the variable `r` and the variable `e` it evaluates the expression `exp2` and assigns the new determined value again to the variable `r`. This process is repeated until all elements of the collection have been assigned to variable `e`.
 It is unclear whether it makes sense to apply the `iteration` operator to a partial expression which either returns a single element or the undefined value. The rule introduced here tolerates such a situation based on the assumption that partial expressions and sequences are treated similarly.
6. $\text{Type}[\text{if } b \text{ then } e1 \text{ else } e2 \text{ endif}] := (S, T)$, where
 $T = \text{scs}*(\text{Type}[e1].\text{Name}, \text{Type}[e2].\text{Name})$ if
 $\text{Type}[b] = (\text{simple}, \text{Boolean})$
 $S = \text{Type}[e1].\text{Sort}$ if $\text{Type}[e2].\text{Sort}$ compatible with $\text{Type}[e1].\text{Sort}$
 $S = \text{Type}[e2].\text{Sort}$ if $\text{Type}[e1].\text{Sort}$ compatible with $\text{Type}[e2].\text{Sort}$
 $S = \text{collection}$ otherwise
 The OCL standard states that the type of the result of an `if-then-else` expression is the type of the subexpression of its then-branch *and* its else-branch. It does not explain the meaning of the word “and” in this case. The type checking rule above chooses a rather liberal interpretation of the word “and” where any combination of sorts of subexpressions `e1` and `e2` is permitted. The sort of the resulting expression is `collection` in the worst case. The most restrictive interpretation of the word “and” would require that $\text{Type}[e1].\text{Sort} = \text{Type}[e2].\text{Sort}$.
7. $\text{Type}[e.\text{oclIsTypeOf}(T\text{Name})] := (\text{simple}, \text{Boolean})$, where
 $T\text{Name}$ conforms to $\text{Type}[e].\text{Name}$ and
 $T\text{Name} \in \text{Classifier}$ and $\text{Type}[e].\text{Sort} = \text{simple}$
 The operator `oclIsTypeOf` may be applied to any expression which returns a single object. It checks whether this object is a direct *instance* of the given type `TName` and returns the appropriate boolean result.

8. $\text{Type}[e.\text{oclIsKindOf}(\text{TName})] := (\text{simple}, \text{Boolean})$, where
 $\text{gcs}*(\text{Type}[e].\text{Name}, \text{TName}) \neq \text{OCLNil}$ and
 $\text{Type}[e].\text{Name}$ does not conform to TName and
 $\text{TName} \in \text{Classifier}$ and $\text{Type}[e].\text{Sort} = \text{simple}$

The operator `oclIsKindOf` may be applied to any expression which returns a single object. It checks whether this object is a *member* of the given type TName and returns the appropriate boolean result. It makes therefore no sense to write down an `oclIsKindOf` expression where static type analysis is already able to guarantee that the computed object has a type which conforms to TName .

9. $\text{Type}[e.\text{oclAsTypeOf}(\text{TName})] := (\text{partial}, T)$, where
 $T = \text{gcs}*(\text{Type}[e].\text{Name}, \text{TName})$ if
 $\text{Type}[e].\text{Name}$ does not conform to TName and if
 $\text{TName} \in \text{Classifier}$ and $\text{Type}[e].\text{Sort} = \text{simple}$

The operator `oclAsTypeOf` may be applied to any expression which returns a single object. It returns the given object if it is a member of the type TName it returns the undefined value otherwise. It makes therefore no sense to write down an `oclAsTypeOf` expression where static type analysis is already able to guarantee that the computed object has a type which conforms to TName .

Note that the type checking rules introduced above do not cover all possible combinations of the types of the subexpressions of a regarded expression. It is a matter of debate whether a default rule should be added that “fires” if no other type checking rules are applicable and assigns the type (\dots, OCLNil) to a subexpression with an otherwise undefined type. We have made the experience that the introduction of such a default type, which is compatible with any other type, allows us to type check expressions with ill-defined subexpressions without generating too many misleading error messages. In any case, a type checking tool has to generate an error message if it encounters an expression with an undefined type or with a type equal to (\dots, OCLNil) .

7 Conclusions and Future Work

In the previous sections we discussed some OCL type checking problems and presented three different solutions for these problems. The first solution relies on the construction of type hierarchies where any pair of types possesses (at most) one smallest common supertype and (at most) one greatest common subtype. Furthermore, it assumes that objects are direct instances of a single type (class). The second solution avoids these restrictions. It uses sets of types as extensional representations for missing uniquely defined supertypes or subtypes. Due to the fact that its accompanying type checking rules are too complex to be useful in practice a third solution has been developed. It makes no assumptions concerning the structure of a regarded type hierarchy and computes a single type for any legal OCL expression (instead of a set of types).

Compared with the currently valid type checking rules of the OCL standard, the proposed new rules have the following advantages:

- They accept reasonable expressions such as $S1 \rightarrow \text{union}(S2)$, where the type of $S1$ conforms to the type of $S2$ but not vice versa which are rejected nowadays.
- They reject useless expressions such as $S1 \rightarrow \text{intersection}(S2)$ or $S1 = S2$ where the type of $S1$ and the type of $S2$ do not possess a common subtype which are accepted nowadays.
- They compute more specific types for collection manipulating expressions such as $S1 \rightarrow \text{intersection}(S2)$ where the type of $S2$ conforms to the type of $S1$ (the new rules return the type of $S2$ instead of the type of $S1$).

However, many important questions concerning the OCL standard version 1.3 still have to be addressed:

- The dynamic semantics of partially defined expressions is more or less undefined.
- The same is true for the transitive closure of associations if the regarded association is not a partial order.
- The available means for reflection (operations which determine the type of an object, the attributes of a type, etc.) do not possess any precisely defined type checking rules.
- ...

It is the subject for future work to fix the dynamic semantics definition of OCL and to write down a complete set of OCL type checking rules based on the proposed OCL meta model in [14]. These new type checking rules should become an accepted part of the forthcoming OCL standard version 2.0. Furthermore, they should be implemented as part of already existing OCL toolkits such as the Dresden OCL compiler [11] or the OCL toolkit from Bremen [15].

References

1. H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 19(2):106–190, 1987.
2. H. Ait-Kaci and R. Nasr. Integrating data type inheritance into logic programming. In M.P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 121–136. Springer Verlag, Berlin, 1989.
3. L. Cardell and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
4. T. Clark. Typechecking UML static model. In [9], pages 503–517, 1999.
5. St. Cook, A. Kleppe, and R. Mitchell et al. The Amsterdam manifesto on OCL. Technical report, 2000. <http://www.trireme.com/amsterdam/manifesto-1-5.pdf> (visited: 11/07/2000).

6. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.
7. A. Evans and St. Kent, editors. *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, volume 1939 of *Lecture Notes in Computer Science*, Berlin, 2000. Springer Verlag.
8. UML Revision Task Force. OMG unified modeling language specification v. 1.3, document ad/99-06-08. Technical report, Object Management Group, 2000. <http://www.omg.org/uml/> (visited: 07/11/2000).
9. R. France and B. Rumpe, editors. *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, Berlin, 1999. Springer Verlag.
10. Martin Hitz and Gerti Kappel. *UML@Work - Von der Analyse zur Realisierung*. dpunkt.lehrbuch. dpunkt.verlag, Heidelberg, 1998.
11. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In [7], pages 278–293, 2000.
12. Jörg-Volker Müller. *Entwurf einer objektorientierten Programmiersprache mit statischem Typkonzept und Parallelität*. Shaker Verlag, Aachen, 1994.
13. The precise UML group. PUML home page. Technical report, 2000. <http://www.cs.york.ac.uk/puml/> (visited: 11/07/2000).
14. M. Richters and M. Gogolla. A metamodel for OCL. In [9], pages 156–171, 1999.
15. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In 9, pages 265–277, 2000.
16. Andy Schürr. Adding graph transformation concepts to uml's constraint language ocl. In J. Padberg, editor, *appears in: Proc. UNIGRA Satellite Workshop of ETAPS 2001*, Amsterdam, 2001. Elsevier Science Publ.
17. Andy Schürr. New type checking rules for OCL (collection) expressions. 2001.
18. Andy Schürr, Andreas J. Winter, and Albert Zündorf. PROGRES: Language and environment. In [6], pages 487–550. 1999.
19. J. Warmer and A. Kleppe. *OCL: The Object Constraint Language - Precise Modeling with UML*. Addison Wesley, New York, 1999.