

# New Type Checking Rules for OCL Expressions

Andy Schürr

Institute for Software Technology  
University of the Federal Armed Forces, Munich  
D-85577 Neubiberg, Germany  
e-mail: Andy.Schuerr@unibw-muenchen.de

**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 (enough) defined in the UML standard version 1.3. Furthermore, they have certain deficiencies concerning the treatment of collection manipulating operations. This paper sketches three different approaches for the definition of modified OCL type checking rules. These proposals are based on our experiences with the design of a rather similar constraint language that is part of the graph transformation language PROGRES.

## 1 Introduction

The *object constraint language OCL* was from the very beginning an integral part of OMG's versions 1.x of the UML standard. Nevertheless, many UML books do not explain OCL in detail or even neglect the existence of OCL completely. In its current form OCL (version 1.3) suffers from the same problems as many parts of the UML standard: it neither possesses a precise static semantics nor a precise dynamic semantics definition [UML00]. These are the reasons, why groups of researchers are now active to refine and redesign parts of OCL in order to influence the UML 2.0 definition process [AN89], [Co00], [PUM00].

It is the purpose of this paper to apply our experiences with the development of the graph transformation language PROGRES to OCL. PROGRES is a visual, executable specification language that combines a subset of UML class diagrams for the definition of graph schemata with graph transformation rules for the definition of object structure manipulations and with OCL-like path expressions for the definition of integrity constraints and complex graph queries [Schü97], [SWZ97].

The PROGRES *path expression sublanguage* is similar to OCL with respect to the following properties:

- It is related to UML-like class diagrams in the same way as OCL.
- It combines the usual expressions for the manipulation of boolean values, strings, integers, and so forth with path expressions for navigation along associations, too.
- It distinguishes between partially defined and always defined path expressions as well as between single object returning and collection (set) returning path expressions.

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

- (1) Our path expressions have a well defined set of type checking rules expressed as predicate logic formulas.
- (2) Furthermore, PROGRES has a precisely defined, but nevertheless still rather abstract operational 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.

In the following we will focus our interest on topic (1) above, i.e. the construction of appropriate *OCL type checking rules* based on our experiences with the formal definition of the PROGRES language, the vast body of knowledge about type checking polymorphic specification and programming languages in general [AN89], [CW85], and a recently published paper about the OCL language's type system [C199]. This paper presents for the first time a precise definition of the OCL type checking rules, but excludes the problems addressed here concerning operations on collections (sets) of different types.

The discussion of the addressed type checking problems is organized as follows: In the following Section 2 we explain the problems with the currently circulating version of the OCL type checking rules for collection operators like union, intersection, and includesAll. Furthermore, we show that variants of this problem affect the type checking rules for the comparison of objects or collections of objects, too. Section 3 presents the PROGRES solution for these problems, which follows the lines of the type checking approach invented for the logic-based language LOGIN [AN89]. It relies on the fact that our 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 (Any or Nil in the worst case). Section 4 presents a slightly different solution for the presented type checking problems, which works with the powerset of all OCL types and avoids thereby the restriction of class hierarchies to lattices.

Both solutions of the type checking problem are closely related to each other due to the fact that it is always possible to transform a given class hierarchy into a (for the programmer hidden) lattice, which can be used for type checking purposes [Ai87]. The additional classes of this lattice represent the needed nonsingleton sets of the second type checking approach. Unfortunately, both approaches are too complex for the average OCL user, who does not want care about class lattices or sets of types.

Therefore, Section 5 presents yet another solution, which (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 (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 these type checking rules constitute the proper compromise between the restrictiveness and simplicity of the type checking rules of the OCL standard on one hand and the expressiveness and complexity of the solutions of Section 3 and 4 on the other hand. They should be incorporated into OCL version 2.0 and become part of the currently existing OCL processing tools as explained in Section 6 of this paper.

## 2 Problems with Type Checking OCL Expressions

In the following we show that the implicitly defined OCL version 1.3 type checking rules in [UML00] often return unwanted results and should be changed in the future version 2.0.

For this purpose 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 `NameExpr` computes an object which is a member (direct or indirect instance) of the type `Name`, and that an OCL expression `NameSEExpr` computes a set of members of the type `Name`.

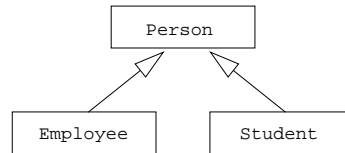


Fig. 1: A simple UML class hierarchy

The OCL standard version 1.3 with its template-like declarations of predefined operations such as

`set->union(set2 : Set(T)): Set(T)` and `set->intersection(set2 : Set(T)): Set(T)`

probably requires that the type of the OCL expressions

`XSEExpr->union(YSEExpr)` and `XSEExpr->intersection(YSEExpr)`

is `set(X)` if `Y` is a subtype (subclass) of `X` and undefined otherwise. This has the consequence that the OCL expressions

`EmployeeSEExpr->union(StudentSEExpr)` and

`EmployeeSEExpr->intersection(PersonSEExpr)`

are illegal, whereas

`EmployeeSEExpr.oclAsType(Person)->asSet->union(StudentSEExpr)` and

`PersonSEExpr->intersection(EmployeeSEExpr)`

are legal expressions of type `set(Person)`. This interpretation of the template-like declaration of OCL standard operations has the following unwanted consequences:

- `PersonSEExpr->union(StudentSEExpr)` and `StudentSEExpr->union(PersonSEExpr)` have different types despite of the fact that set union is a commutative operation,
- `set(EmployeeExpr)` is not the type of `PersonSEExpr->intersection(EmployeeSEExpr)`, and
- `EmployeeSEExpr->includes(StudentExpr)` and `EmployeeExpr = StudentExpr` are legal boolean expressions (the standard requires that the type of the second argument of these expressions is of type `OCLAny`, i.e. allows any element), whereas
- `EmployeeSEExpr->includesAll(StudentSEExpr)` and `EmployeeSEExpr = StudentSEExpr` are illegal boolean expressions.

As a consequence, the standard type checking rules are *not complete* with respect to the OCL language's dynamic semantics as required in [CI99]. Some reasonable OCL expressions with a well-defined dynamic semantics are illegal with respect to the standard type checking rules. Even worse the given rules often return types which are too general, and they return different results for expressions which obviously have the same dynamic semantics.

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

$XSEpr \rightarrow someOperation(YSEpr)$  with  $someOperation \in \{union, intersection, \dots\}$   
the existence of a (smallest) common supertype of  $X$  and  $Y$ , which is then used as parameter  $T$  of the involved operation template

$set \rightarrow someOperation(set2 : Set(T)) : Set(T)$ .

Such a smallest common supertype always exists due to the fact that all non-collection types are subtypes of  $OclAny$ . 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.

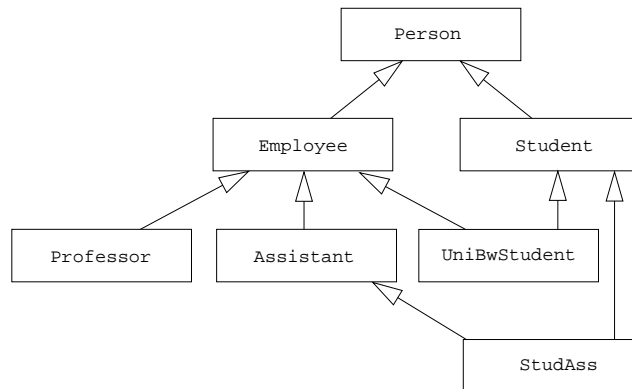


Fig. 2: Example of a ill-designed but legal UML class hierarchy

Fig. 2 presents one example of an UML class diagram that uses multiple inheritance. Both  $Stud(ent)Ass(istants)$  and  $UniBwStudents$  are subtypes (subclasses) of  $Employee$  and  $Student$ . Probably such a class diagram is ill-designed, but it is legal. Referring to 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

- (1)  $StudAssSEpr \rightarrow union(UniBwStudentSEpr)$
- (2)  $EmployeeSEpr \rightarrow intersection(StudentSEpr)$

Using the new "smallest common supertype" rule we have to conclude that

- the union of a  $StudAssSEpr$  and a  $UniBwStudentSEpr$  has not a single, but two smallest common supertypes  $set(Employee)$  and  $set(Student)$ ,
- whereas the intersection of an  $EmployeeSEpr$  and a  $StudentSEpr$  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  $set(Person)$ . 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, which could be recognized at compile time. Consider for instance the expressions

- (3) ProfessorSEpr->intersection(AssistantSEpr)
- (4) ProfessorSEpr - AssistantSEpr<sup>1</sup>
- (5) ProfessorSEpr = AssistantSEpr

We know from the class diagram of Fig. 2 that the first expression always returns the empty set, that the second expression always returns the collection computed by its subexpression ProfessorSEpr, and that the third expression always returns the result false.

### 3 Type Checking OCL Expressions - the Lattice Solution

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

Therefore, the class diagram of Fig. 2 is illegal, which is recognized by an incremental *lattice checking algorithm*. An automatically working *completion algorithm* adopted from [Ai87] transforms the class diagram of Fig. 2 into the class diagram of Fig. 3 by adding a single class EmployeeStud (cf. [SWZ97] for further details).

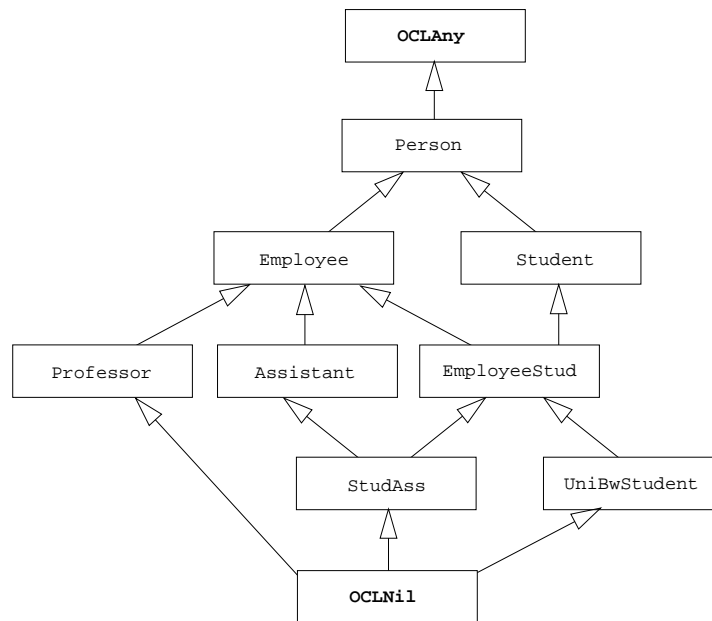


Fig. 3: Redesigned class hierarchy with lattice property

Relying on the automatically constructed class diagram of Fig. 3 we can introduce the following OCL type checking rules:

---

<sup>1</sup> The "-" operator defines the difference between two sets of elements. It is a matter of debate whether a future OCL version should not use a syntax for set difference which is more similar to the syntax for set union and set intersection.

- (1)  $\text{Type[ expr1 \rightarrow \text{union}(expr2) ]} := \text{scs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]})$ ,  
if  $\text{scs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]}) \neq \text{set}(\text{OclAny})$
- (2)  $\text{Type[ expr1 \rightarrow \text{intersection}(expr2) ]} := \text{gcs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]})$ ,  
if  $\text{gcs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]}) \neq \text{set}(\text{OclNil})$
- (3)  $\text{Type[ expr1 \rightarrow \text{excluding}(expr2) ]} := \text{Type[expr1]}$ ,  
if  $\text{gcs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]}) \neq \text{set}(\text{OclNil})$
- (4)  $\text{Type[ expr1 = expr2 ]} := \text{boolean}$ ,  
if  $\text{gcs}(\text{Type[ expr1 ]}, \text{Type[ expr2 ]}) \neq \text{set}(\text{OclNil})$

with  $\text{scs}(t, t')$  being the smallest common supertype of  $t$  and  $t'$  and with  $\text{gcs}(t, t')$  being the greatest common subtype of  $t$  and  $t'$ . The function  $\text{scs}$  returns the type  $\text{OclAny}$  (or  $\text{set/bag/sequence}(\text{OclAny})$  for sets, bags, and sequences) if the common supertype does not exist; the function  $\text{gcs}$  returns  $\text{OclNil}$  (or  $\text{set/bag/sequence}(\text{OclNil})$  for sets, bags, and sequences) if the common subtype does not exist.

Applying these rules to the class diagram of Fig. 3 and the expressions (1) and (2) of Section 2 we get the following results:

$$\begin{aligned} &\text{Type[StudAssSEExpr} \rightarrow \text{union}(\text{UniBwStudentSEExpr})] \\ &= \text{scs}(\text{set}(\text{StudAss}), \text{set}(\text{UniBwStudent})) = \text{set}(\text{scs}(\text{StudAss}, \text{UniBwStudent})) \\ &= \text{set}(\text{EmployeeStud}) \end{aligned}$$

and

$$\begin{aligned} &\text{Type[EmployeeSEExpr} \rightarrow \text{intersection}(\text{StudentSEExpr})] = \text{gcs}(\text{set}(\text{Employee}), \text{set}(\text{Student})) \\ &= \text{set}(\text{gcs}(\text{Employee}, \text{Student})) = \text{set}(\text{EmployeeStud}) \end{aligned}$$

Furthermore, the presented rules reject the expressions (3) through (5) of Section 2 due to the fact that Professor and Assistant have no (greatest) common subtype except of  $\text{OclNil}$ .

The type checking rules for other OCL operators have a similar form. The main drawback 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). Often these requirements enforce the construction of better designed class hierarchies. Nevertheless, our experiences show that it is often 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, which manipulate sets of OCL types instead of relying on the lattice property of class hierarchies.

#### 4 Type Checking OCL Expressions - the Type Set Solution

One solution for avoiding the lattice restriction for type (class) 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 [Ai87]. Whenever the type checking algorithm computes one of these intermediate types for a given OCL expression, it displays its set of smallest user defined superclasses. 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*.

The appropriate type checking rules, which compute sets of OCL types, have the following form for the operators union and intersection:

- (5)  $\text{Type}[ \text{expr1} \rightarrow \text{union}(\text{expr2}) ] :=$   
 $\text{rmSupertypes}(\text{allSupertypes}(\text{Type}[ \text{expr1} ])) \cap \text{allSupertypes}(\text{Type}[ \text{expr2} ]),$   
 $\text{Type}[ \text{expr1} \rightarrow \text{union}(\text{expr2}) ] \neq \emptyset$
- (6)  $\text{Type}[ \text{expr1} \rightarrow \text{intersection}(\text{expr2}) ] :=$   
 $\text{rmSupertypes}(\text{allSupertypes}(\text{Type}[ \text{expr1} ])) \cup \text{allSupertypes}(\text{Type}[ \text{expr2} ])$

where the used auxiliary functions are defined as follows (using OCL):

```
allSupertypes(set) = set->union(directSupertypes.allSupertypes)
-- allSupertypes 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
rmSupertypes(set) = set->reject( directSubtypes->intersection(set)->notEmpty )
-- rmSupertypes 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 [StudAssSEExpr->union(UniBwStudentSEExpr)] =
rmSupertypes({set(StudAss),set(Assistant),set(Employee),set(Student), set(Person))} ∩
{set(UniBwStudent),set(Employee),set(Student), set(Person)})
= rmSupertypes( {set(Employee), set(Student), set(Person)} )
= {set(Employee),set(Student)}
Type[ EmployeeSEExpr->intersection(StudentSEExpr) ]
= rmSupertypes( {set(Employee), set(Person)} ∪ {set(Student),set(Person)} )
= {set(Employee),set(Student)}
```

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 of the fact that UML allows models, where one object is a direct instance of more than one type (class). 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 the depicted situation so that it is possible to re-introduce the `oclType` operator together with other means for reflection.

The fact that objects may be direct instances of more than one type (class) had an important impact on the definition of the type checking rule (2') above. This rule does not make any attempts to compute a set of greatest common subtypes of the types of the regarded sub-expressions. Therefore, the computed type set of

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

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

## 5 Type Checking OCL Expressions - the Simple Solution

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 the following properties:

- The new type checking rules accept more expressions than the standard rules and they return in certain cases more specific types than the standard rules.
- The new rules return for any legal expression a single type which is a supertype of the type of its computed result (if any regarded object is a direct instance of a single 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 (OCLAny in the worst case).

Applying the new type checking rules to the OCL expressions (1) and (2) of Section 3 we expect therefore the following results for the class hierarchy of Fig. 2:

Type [StudAssistantSEExpr->union(UniBwStudentSEExpr)] = set(Person)

Type[EmployeeSEExpr->intersection(StudentSEExpr)] = set(Person)

Due to lack of space we cannot present the complete set of modified type checking rules for all OCL operators nor prove that the modified rules fulfill indeed the above listed requirements. To give the reader nevertheless an impression how the needed modifications look like we conclude this section with a presentation of the redesigned rules for the operators union, intersection, excluding, and equality for collections:

(1'') Type[ expr1->union(expr2) ] := up(scs( {Type[ expr1 ], Type[ expr2 ] } ))

(2'') Type[ expr1->intersection(expr2) ] := up(gcs( {Type[ expr1 ], Type[ expr2 ] } )),  
if gcs( {Type[ expr1 ], Type[ expr2 ] } )  $\neq \emptyset$

(3'') Type[ expr1->excluding(expr2) ] := Type[expr1],  
if gcs( {Type[ expr1 ], Type[ expr2 ] } )  $\neq \emptyset$

(4'') Type[ expr1 = expr2 ] := boolean, if gcs( {Type[ expr1 ], Type[ expr2 ] } )  $\neq \emptyset$

The needed auxiliary functions are defined as follows:

up( TSet ) := if TSet = { t } then t else up(scs(TSet))

scs(TSet) := { t  $\in$  OCLType |  $\forall t_s \in$  TSet:  $t_s$  conformsTo t  $\wedge$   
 $\neg \exists t' \in$  TSet:  $\forall t_s \in$  TSet:  $t_s$  conformsTo t'  $\wedge$   
t  $\neq$  t'  $\wedge$  t' conformsTo t }

gcs(TSet) := { t  $\in$  OCLType |  $\forall t_s \in$  TSet: t conformsTo t\_s  $\wedge$   
 $\neg \exists t' \in$  TSet:  $\forall t_s \in$  TSet: t' conformsTo t\_s  $\wedge$   
t  $\neq$  t'  $\wedge$  t conformsTo t' }

The functions scs and gcs compute the set of smallest common supertypes and greatest common subtypes of a given set of types wrt. the partial order defined by the conformsTo relation, whereas the recursively defined function up replaces a set of types by a single *type approximation*. This approximation is the smallest common supertype of a regarded set of types TSet, which is computed as follows: Applied to TSet the function up either returns

the single element of TSet or calls itself applied to the smallest common supertype set of TSet<sup>2</sup>.

It is quite obvious that the new rules behave like the ones introduced in section 3 as long as our type hierarchies are (semi-)lattices, i.e. as long as `scs` and `gcs` return singleton sets. Furthermore, our running example shows that the modified rules accept OCL expressions which are rejected by the standard type checking rules and that they compute more specific types under certain circumstances. Furthermore, it is worth-while to notice that the new rules require the existence of the greatest type `OCLAny`, but that they do not require the existence of the smallest type `OCLNil`. All calls to function `gcs`, which return the empty set of types (instead of `OCLNil`), correspond to forbidden situations, where static analysis is already able to determine the result of the regarded expression:

- The intersection of two sets is the empty set if their static types do not possess a common subtype.
- The comparison of two elements or sets of elements yields `false` if their static types do not possess a common subtype.
- The exclusion of a set `S1` from another set `S2` is always equal to `S2` if the static types of `S1` and `S2` do not possess a common subtype.

The formal definition of this last variant of OCL type checking rules seems to be rather complex compared with the rules of section 3. Nevertheless, we believe that the basic idea behind the definition of these rules is quite intuitive so that the average OCL user will have no problems to understand the generated error messages.

## 6 Conclusions and Future Work

In this paper we have discussed one category of OCL type checking problems and presented three different solutions for the problem. 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. 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->union(S2)`, where the type of `S1` conforms to the type of `S2` but not the other way round, which are rejected nowadays.
- They reject useless expressions such as `S1->intersection(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 expressions such as `S1->intersection(S2)`, where the type of `S2` conforms to the type of `S1` (`S2` instead of `S1`).

---

<sup>2</sup> Computations of function up obviously terminate as long as we are dealing with finite type hierarchies, where any `t1 conformsTo t2 conformsTo ... conformsTo tn` chain is finite.

Nevertheless, many other questions concerning the OCL standard version 1.3 still have to be addressed. This includes e.g. the fact that operators for the definition of transitive closures, default values for partially defined expressions, selection of elements from singleton sets, etc. are missing. Furthermore, it would be possible to define more sophisticated type checking rules which keep track of lower and upper boundaries for collection computing subexpressions, distinguish between object level and type level expressions, and take care of nondeterministic selections of set elements, one of the suggested extensions for OCL version 1.4 [K100]. It is the subject for future work to write down such a complete set of OCL type checking rules based on the proposed OCL meta model in [RG99]. These new type checking rules should become part of already existing OCL toolkits such as the Dresden OCL compiler [HDF00] or the OCL toolkit from Bremen [RG00].

## References

- [Ai87] H. Ait-Kaci, R. Boyer, P. Lincoln, R. Nasr: Efficient Implementation of Lattice Operations, in: ACM Transactions on Programming Languages and Systems, vol. 19, no. 2, ACM Press (1987), pp. 106-190
- [AN89] H. Ait-Kaci, R. Nasr: Integrating Data Type Inheritance into Logic Programming, in: M.P. Atkinson, P. Buneman, R. Morrison (Eds.): Data Types and Persistence, pp. 121-136, Springer Verlag (1989)
- [Ar00] ArgoUML Web Site, <http://argouml.tigris.org/index.html> (visited: 07/11/2000)
- [CW85] L. Cardelli, P. Wegner: On Understanding Types, Data Abstraction and Polymorphism, in: ACM Computing Surveys, vol. 17, no. 4, pp. 471-522, ACM Press (1985)
- [CI99] T. Clark: Typechecking UML Static Model, in: R. France, B. Rumpe (Eds.): Proc. 2nd Int. Conf. Unified Modeling Language (UML'99), LNCS 1723, pp. 503-517, Springer Verlag (1999)
- [Co00] St. Cook, A. Kleppe, R. Mitchell et al: The Amsterdam Manifesto on OCL, <http://www.trireme.com/amsterdam/manifesto-1-5.pdf> (visited: 11/07/2000)
- [HDF00] H. Hussmann, B. Demuth, F. Finger: Modular Architecture for a Toolset Supporting OCL, in: A. Evans, St. Kent (Eds.): Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), LNCS 1939, pp. 278-293, Springer Verlag (2000)
- [RG99] M. Richters, M. Gogolla: A Metamodel for OCL, in: R. France, B. Rumpe (Eds.): Proc. 2nd Int. Conf. Unified Modeling Language (UML'99), LNCS 1723, pp. 156-171, Springer Verlag (1999)
- [RG00] M. Richters, M. Gogolla: Validating UML Models and OCL Constraints, in: A. Evans, St. Kent (Eds.): Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), LNCS 1939, pp. 265-277, Springer Verlag (2000)
- [K100] Klasse Objecten : OCL 1.4 - The upcoming UML 1.4 version, <http://www.klasse.nl/ocl/> (visited: 07/11/2000)
- [Na96] M. Nagl (ed.): Building Tightly Integrated Software Development Environments, LNCS 1170, Springer Verlag (1996)
- [PUM00] The precise UML group: PUM Home Page, <http://www.cs.york.ac.uk/puml/>, (visited: 07/11/2000)
- [Schü97] A. Schürr: Programmed Graph Replacement Systems, in: G. Rozenberg (ed.), Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations, pp. 479-546, World Scientific (1997)
- [SWZ97] A. Schürr, A. Winter, and A. Zündorf: PROGRES: Language and Environment; in: H. Ehrig, G. Engels, H-J. Kreowski, G. Rozenberg (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Specification and Programming, pp. 487-550, World Scientific (1997)
- [UML00] UML Revision Task Force: OMG Unified Modeling Language Specification v. 1.3. document ad/99-06-08, Object Management Group (1999), <http://www.omg.org/uml/> (visited: 07/11/2000)
- [WK99] J. Warmer, A. Kleppe: OCL: The Object Constraint Language - Precise Modeling with UML, Addison Wesley (1999)