Internal Repport
Number: IRIT/RR--2017--02--FR

A Dynamic Type System for OCL

Thierry Millan, Hervé Leblanc, Christian Percebois
Paul Sabatier University - IRIT, macao team, Université Paul Sabatier,
118 route de Narbonne, F-31062 Toulouse Cedex 4, France

The OCL language is now well-accepted by the modeling community. To be compliant with the regularly updated standard, evaluators must ensure that OCL is a strongly typed language. Indeed, during the analysis of a rule, the evaluator must ensure a correct and a deterministic type of each sub-expression handled. We have chosen to provide a dynamic type system for our OCL evaluator. Moreover, formalizing the OCL type system has the advantage of helping to clarify the standard. This formalization uses the natural deduction logic to represent typing rules and was implemented throughout a type synthesizer in a platform named Neptune.

Key Words: OCL, dynamic type system synthesis, natural deduction logic

1. Introduction

The OCL (Object Constraint Language) language is a key component of the combined products that are dedicated to modeling, and is promoted by the OMG. Firstly, it is used to formalize constraints on UML (Unified Modeling Language) models at M1 or M2 level and checked respectively at M0 or M1 level. Constraints expressed at M1 level allow the checking of both structural and behavioral constraints on a user model. Constraints expressed at M2 level define WFR (well-formedness rules) at a metamodeling level to establish conformance relationships between a model and its metamodel. Currently, three tools are emerging: Dresden OCL [Wilke et al. 2011], USE [Gogolla et al. 2007] and MDT OCL [Garcia and Shidqie 2007] included in Eclipse OCL. These tools took the latest releases of the OCL standard into account in their implementations [OMG 2010] and are regularly updated [OMG 2012]. OCL is also used for model transformation in the Queries/Views/Transformations (QVT) specification [OMG 2011a]. For example, the ATL language [Jouault and Kurtev 2006; Bruneliere et al. 2015] is based on this standard for both its data types and its declarative expressions.

To be compliant with the standard, tools have to take into account that OCL is a strongly typed language. Jones and Liskov establish that a strongly typed language is a language in which each type defines a set of primitive operations that are the only way to act on objects of this type [Jones and Liskov 1978]. The OCL specification agrees with this definition and suggests the use of a type checker:

> "A type is assigned to every OCL expression and typing rules determine in which ways well-formed expressions can be constructed" [OMG 2010, p.191].

Indeed, the evaluator must ensure a correct type for each sub-expression handled during the analysis of an OCL rule. Using a strongly typed language imposes having a complete and rigorous language specification, and an analysis policy for computation of the correct type for a rule. Indeed, a designer of a new domain specific modeling language mainly focus on the expressiveness of the language i.e. metaclasses and their relationships, and some restrictions expressed by OCL rules. In this context, dynamic typing on OCL rules allows reducing development costs and offers more flexibility to manage changes during the design of a new modeling language from scratch [Tratt 2009].

Concerning the first point, the OCL standard contains some inaccuracies and inconsistencies. An example of one such an inconsistency concerns the definition of the *OclAny* type. First, the standard indicates that all types conform to *OclAny* [OMG 2010, p.139]), but claims also [OMG 2010, p.205]) that *OclAny* is not the supertype of collections. Another example of imprecision concerns the definition domain of the primitive types *Integer* and *UnlimitedNatural*. For example, the evaluation of expression (2 > *) gives false if 2 is considered as an *unlimitedNatural* or returns invalid if 2 is considered as an *integer* [Willink 2006]. This problem arises because the intersection of the definition domains of the two primitive types is not empty and because it lacks a notation to precise the type of a constant.

The second point concerns the analysis policy for the computation of the correct type of an expression in the context of a strongly typed language. The type checker can be implemented either as a static or a dynamic type system. As OCL is a strongly typed language, the programmer must ensure the correct typing of each sub-expression of an OCL rule and then must use specific type constructors and cast operators. The three tools

mentioned above and the studies carried out in [Schürr 2002; Cengarle and Knapp 2004; Kyas 2005] have chosen a static typing policy.
Unlike all these approaches, in this work we have chosen to provide a dynamic typing policy for the OCL language. This requires a type synthesizer which computes and provides types instead of the programmer. We have chosen to facilitate the design of OCL rules for programmers over providing the feedback that the evaluation will be successful in all cases. Type checks are more accurate because they use dynamic information as well as coding information. As is the case in other works, we must ensure that the typing of each interpreted expression is valid and deterministic. In order to achieve this aim, we formalize the OCL type system: OCL types are represented by partially ordered sets (posets) and the type checking during the expression analysis by the natural deduction logic formalism. Our type synthesizer was implemented in a platform named Neptune, which includes an OCL evaluator [Millan et al. 2009] dedicated solely to the verification of WFR expressed at M2 level. We have only focused on OCL-MOF subset [OMG 2012]. In the metamodeling stage, modelers may need to develop OCL rules for checking the conformance between models and metamodels. At this stage modelers only need a language and a flexible tool to allow the swift writing and debugging of OCL rules.

The rest of this paper is organized as follows: In Section 2 we present some examples of expressions illustrating the necessity to provide a dynamic type checker for the OCL language. Section 3 presents the OCL type hierarchy and details the two required operations: *isSubtypeOf* and *smallestCommonSuperType*. Section 4 presents principal typing rules using natural deduction logic. Section 5 is dedicated to our OCL platform that we named Neptune. Section 6 is dedicated to related work. A discussion about the results of experiments and some omissions in the standard follows in Section 7. Finally, a conclusion summarizes our approach and explains future activities on this topic.

2. The need for a dynamic type checker

A type checker can be implemented as a static type system, a dynamic type system [Pierce 2002], or a mix of both. A static type system evaluates the type of an expression during compile-time in order to provide a feedback to the developer that the evaluation will be successfull in all cases. However, this approach is conservative because it rejects some programs that may be well-behaved at runtime, but that cannot be statically well-typed [Tratt 2009]. In addition, statically typed languages require operators in order to modify types at runtime. Even if static typing increases security and trust for designing rules, the excessive use of cast operators weakens security at runtime and increases the complexity of the rules. Even if using cast operators can be used by developers to make explicit some assumptions about the runtime type of an expression, we prefer that developers only focus on the logic of the rule.

A dynamic type system computes the type of an expression during runtime. Potentially, it can be more accurate than a static system, as it uses dynamic information as well as any information from the code. As related in the Caml Light language [Mauny 1995]:

> "Type synthesis may be seen as a game. When learning a game, we must learn the rules, what is allowed, and what is forbidden, and learn a winning algorithm. In type synthesis, the rules of the game are called a type system, and the winning strategy is the type checking algorithm."

Runtime checks only assert that conditions hold in a particular execution of the program, and checks are repeated for every execution. The dynamic typing allows developers to omit casts and thus to become unaware of potential problems. However, in an interpreted setting, developers formulate statically wrong constraints that in some specific contexts evaluate well. Some kinds of programs are, in fact, considerably more difficult to write in a statically typed language than in a dynamically typed one.

For complex models checking rules, we preconize an optimistic way concerning evaluation of the rules. We do not wish that programmers overspecify contraints on types that can never appear. In addition, our OCL interpreter offers a static typing policy on the fly. The following examples illustrate the advantages of using a dynamic type system for evaluating OCL rules.

2.1. The context

In the specification, the relationship binding the different elements is irrespectively precised *is subtype of* or *conforms to*. In the following, we choose the term *is subtype of* most appropriately for languages. Take for example the OCL type hierarchy deduced from our comprehension of the OCL standard (standard and its annex) [OMG 2012] in Figure 1.

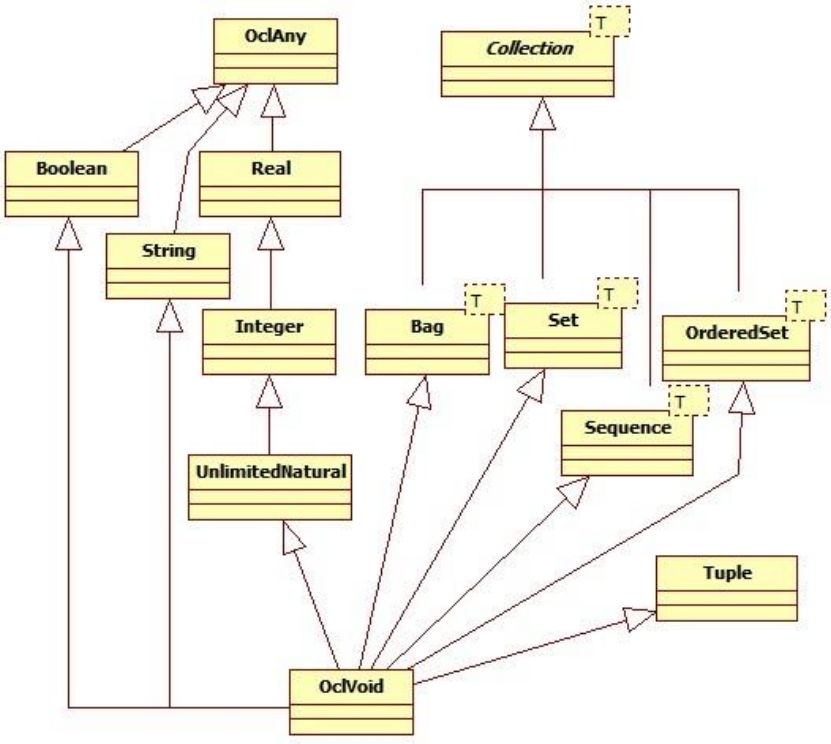

Fig. 1. The OCL type hierarchy.

The OCL type hierarchy is composed of three hierarchies. The first one contains the primitive types *Boolean*, *Real*, *Integer*, *UnlimitedNatural*, and *String*. *UnlimitedNatural* [OMG 2012, p.154] is used to encode the non-negative values of multiplicity specifications. This includes a special unlimited value (*) that encodes the upper value of

a multiplicity. This hierarchy has a common supertype named *OclAny* that provides operations for the handling of all atomic types used in an OCL rule.

The second one represents homogeneous collections. All of these types are parametric types having the element's type as a parameter. *Bag* and *Sequence* can contain elements having the same value whereas *Set* and *OrderedSet* can contain only elements having different values. *Bag* and *Set* are unordered collections whereas *Sequence* and *OrderedSet* are ordered. This hierarchy has a common abstract supertype named *Collection*.

The third one combines different types into a single aggregate type. Parts of a *Tuple* are described by its attributes, each having a name and a type. There is no restriction on the kind of types that can be used as part of a tuple. In particular, a tuple may contain other tuple types and collection types. These three hierarchies have a common subtype named *OclVoid* that is a subtype of all other types [OMG 2012, p.146].

2.2. Executability

After presentation of the predefined types in the standard, we examine the benefits of dynamic typing on OCL expressions using these data types.

Let us now consider this first OCL expression:

Sequence{1, 1.0} *(expr. 1)*

The type of the first element is *Integer* or *UnlimitedNatural* and the type of the second element is *Real*. As it is presented in Figure 1, *Integer* and *UnlimitedNatural* are subtypes of *Real*. Then, the inferred type of expr. 1 is *Sequence(Real)*.

Given now this second expression:

Sequence{1, 1.0}->sum() *(expr. 2)*

Due to the inferred type of expr. 1, the result of expr. 2 is 2.0.

Given a slightly more complex OCL expression:

Sequence{Bag{1}, Set{'a'}, Set{1.0}} *(expr. 3)*

Inferring on the type system presented in Figure 1, we deduce that the type of expr. 3 is *Sequence(Collection(OclAny))*. The *OclAny* type is the first common supertype of *Integer*, *String*, and *Real*. The *Collection* type is the first common supertype of *Bag* and *Set*.

Let us now consider an extended OCL expression of expr. 3:

Sequence{Bag{1}, Set{'a'}, Set{1.0}}->first()->asSequence()->first()+1 *(expr. 4)*

Table I (respectively Table II) presents the analysis step by step of expr. 4 using a static type system (respectively a dynamic type system). For each table, the first column contains the OCL expression and its sub-expressions, the second column gives analysis results and the third type evaluations. Generally, the evaluation of the type and the result of a rule are computed in parallel.

Table I. Analysis step by step of expr. 4 with a static type system.

Expression	Result	Type
Sequence{Bag{1},Set{'a'},Set{1.0}}		Sequence (Collection(OclAny))
->first()	Bag{1}	Collection(OclAny)
->asSequence()	Sequence{1}	Sequence(OclAny)
->first()	1	OclAny
+1	Error	

Table I shows us that an error occurred during a static analysis. In fact, the type of the rule is deduced statically from the sequence parameter. When you attempt to apply the operator +, the static type checker detects an incoherency between the first element typed *OclAny* and the types expected for the operator.

Table II. Analysis step by step of expr. 4 with a dynamic type system.

Expression	Result	Type
Sequence{Bag{1}, Set{'a'},Set{1.0}}		Sequence (Collection(OclAny))
->first()	Bag{1}	Bag(Integer)
->asSequence()	Sequence{1}	Sequence(Integer)
->first()	1	Integer
+1	2	Integer

Table II shows that during a dynamic analysis of the expression, the collection's type is inferred taking into account its elements. When applying the *first* operator, the first element of the collection is extracted and the type system accesses its dynamic type *Bag(Integer).* After a cast to a sequence of integers, the first element is extracted and its type is *Integer*. Then the + operator can be successfully applied.

This example shows that an OCL expression may have a valid result although the static analysis of the rule failed.

2.3. Readability and safety

Let us consider the following basic metamodel with the same declared feature *f* in Figure 2 and the simple OCL expression:

A.allInstances()->union(B.allInstances()).f (expr. 5)

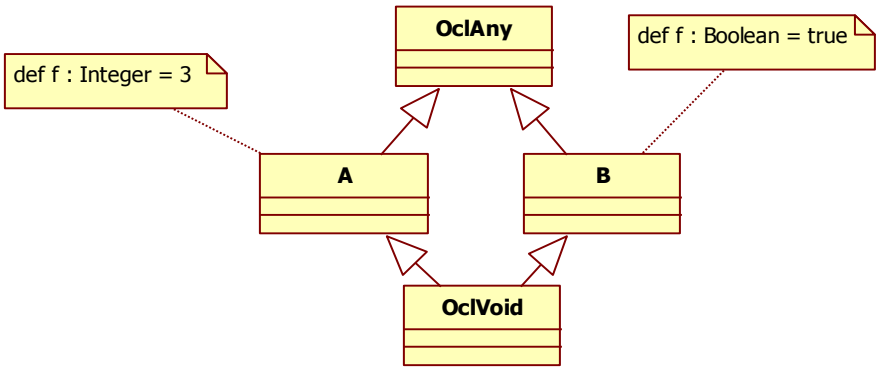

Fig. 2. A metamodel with two different features f.

Two added classes compose this metamodel. Each class owns a pseudo feature *f* that returns respectively an *Integer* for *f* in *A*, and a *Boolean* for *f* in *B*. A pseudo feature is a value or a method added to a metaclass's context using the OCL *def* operation.

If we consider a static typing checking policy, expr. 5 is analyzed as follows:

- the type of *A.allInstances()* is *Set(A)*,
- the type of *B.allInstances()* is *Set(B)*,
- the type of the union set is *Set(OclAny)*,
- and, finally *f* is not a feature of *OclAny* and then *f* is not applicable on each element of the union set.

If we now consider expr. 5 at runtime using a dynamic typing policy, the result is computed without error because after applying the union operation, *f* is applied to each element that has a type supporting the pseudo feature *f*. However, it is possible to rewrite this expression in order to bypass the type checker as follows:

A.allInstances()->union(B.allInstances())-> iterate(i; res = Bag{} |
if i.oclIsKindOf(A) then res->including(i.oclAsType(A).f)
else res->including(i.oclAsType(B).f) endif) *(expr. 6)*

Now, all the typing errors may only be detected during the runtime analysis due to the presence of the operator *oclAsType*. Finally, we obtain the same result as expr. 6 but expr. 5 is more readable and simpler. Taking into account the safety of the two expressions, there is no difference because in both cases the errors occur during runtime analysis.

The presented example is voluntarily basic for ease of understanding. Expr. 5 can be rewritten in a simple manner, applying the *f* operation to the different sets of instances and then computing the union:

A.allInstances().f->union(B.allInstances().f) *(expr. 7)*

However, for complex verification it is sometimes necessary to decorate metaclasses of additional operations. For example we were involved in the DOMINO project (http://www.domino-rntl.org) that proposed a set of concepts and techniques to domain

experts (CNES[1] and Airbus) so that they can define and establish their own model-based processes. One of the case studies was a model transformation from an O2PL procedure to an UML activity diagram [Baudry et al. 2010]. Our aim was to verify that the chain of activities of the UML diagram was homomorphic to the corresponding O2PL procedure. The verification of this homomorphism requires to represent O2PL and activity diagram by abstract syntax trees. To manage the abstract syntax tree of an O2PL procedure, an additional operation named *buildAST* has been implemented for all the non-terminal O2PL grammar. In this case, with a dynamic typing policy the invocation of the right *buildAST* operation is immediate. In addition, the OCL standard does not provide any indication about additional operation evaluation.

2.4. Concrete examples

Between 2005 and 2010 we were involved in the TOPCASED project (http://www.topcased.org). This project aimed to implement an open source MDE framework. The TOPCASED project uses the Eclipse OCL evaluator that includes a static type system. A deliverable [TOPCASED 2010] of this project consists of a report containing several rules written in OCL. Some of these rules concern WFR rules on SAM models [TOPCASED 2005]. SAM is a domain specific language for data flows and synchronous controls dedicated to the functional decomposition and to the behavioral description of avionic components (Airbus). OCL rule 1 checks that the name of a modeled system in Sam is unique.

OCL RULE 1. System unique name using static typing

```
context System inv system_unique_name:
      if (not eContainer().oclIsUndefined()) then
            if (eContainer().oclIsTypeOf(System)) then          ❶
                eContainer().oclAsType(System).listElements.
                   oclAsType(NamedItem)->                       ❷
                   union(eContainer().oclAsType(System).
                       listStorages.oclAsType(NamedItem))->     ❸
                          forAll(e | e <> self  implies e.name <> name)
            else true
            endif
      else true
      endif
```

The second *if* (❶) checks if the result of *eContainer()* is of type *System*. If it is true, we have the guarantee that the result is of type *System*. So the *oclAsType (System)* is only required to have a correct type evaluation at compilation time. The others *oclAsType* (❷, and ❸) seem useless because there are no preliminary checks of the expression types as for the *eContainer* operation. Then we consider that the designer of this rule trusts that the element's type contained in *listElements* is always right. Nevertheless, if an element in *listElements* does not conform to the type *NamedItem* the error will be raised only at runtime as with our dynamic type system. With our approach, we express this rule more simply as follows:

[1] Centre National d'Etudes Spatiales

OCL RULE 2. System unique name using dynamic typing

```
context System inv system_unique_name:
       if (not eContainer().oclIsUndefined()) then
              if (eContainer().oclIsTypeOf(System)) then
                    eContainer().listElements->
                           union(eContainer().listStorages)->
                     forAll(e | e <> self  implies e.name <> name)
              else true
              endif
       else true
       endif
```

In addition, *oclIsTypeOf* and *oclAsType* operators are evaluated during the evaluation of the rule. With these operators the errors occur only at runtime and cannot be thrown during compilation time.

The OCL rule 3 shows the uselessness of the *oclAsType* operator. The first if ensures that the result of *mc*() is of type *System*. The four following *oclAsType* expressions decrease the readability of the rule.

OCL RULE 3. Port is connected with child using static typing

```
context Port inv port_is_connected_with_child:
       if (mc().oclIsTypeOf(System)) then
            mc().oclAsType(System).listFlows->                                  ❶
              select(f |  (f = getInputFlow()) or
                             (f = getOutputFlow()))->notEmpty() or
             (mc().oclAsType(System).listElements->isEmpty() and                ❷
            (mc().oclAsType(System).listGates->isEmpty() and                    ❸
                  (mc().oclAsType(System).listStorages->isEmpty())              ❹
       else true
       endif
```

With our approach we simply express the rule as follows:

OCL RULE 4. Port is connected with child using dynamic typing

```
context Port inv port_is_connected_with_child:
       if (mc().oclIsTypeOf(System))
       then  mc().listFlows->select(f |  (f = getInputFlow()) or
                     (f = getOutputFlow()))->notEmpty() or
              (mc().listElements->isEmpty() and
                (mc().listGates->isEmpty() and
                (mc().listStorages->isEmpty())
       else true
       endif
```

With our approach the designer of the rules only focuses on the logic of the rules. In fact metamodels are heavily based on class hierarchies. It implies that checking conformance models rules excessively use transtyping operators.

3. The OCL type hierarchy and its basic operations

We have chosen to interpret rules expressed at metamodel level, *i.e.*, at M2 level. So, two OCL types (*OclState* and *OclMessage*) are omitted because they have no significance for checking models at M2 level. Before giving rules for the dynamic typing of OCL rules, this section introduces the partially ordered set of types provided by the standard.

Moreover, type checking requires the definition of two operations in order to handle the type hierarchy: *isSubtypeOf* (*iSO*) and *smallestCommonSuperType* (*sCST)*. The *isSubtypeOf* operation corresponds to the partial order relationship in the OCL type hierarchy. For two types *t1* and *t2*, the *commonSuperType(t1, t2)* operation computes a type *t3*, where *t1* and *t2* are both subtypes of *t3*. The *smallestCommonSuperType(t1, t2)* operation ensures that there is no type *t4*, subtype of *t3*, but in which both *t1* and *t2* can be subtypes of *t4*.

3.1. The OCL type hierarchy

In this context, we focus our study on four sets of types:

- the homogeneous collections that we named $\mathcal{P}_c$,
- the heterogeneous collections that we named $\mathcal{P}_t$,
- the primitive types that we named $\mathcal{P}_p$,
- the metaclasses constituting a metamodel that we named $\mathcal{P}_m$.

The *conforms to* relation is defined in [OMG 2010] as:

> "a type type1 conforms to a type type2 when an instance of type1 can be substituted at each place where an instance of type2 is expected. The type conformance rules for types in the class diagrams are simple: each type conforms to each of its supertypes and type conformance is transitive."

In fact, the OCL specification defines the subtype/supertype relationship using a type hierarchy. We confound *conforms to* and *subtype of* relations and we build the full type system ($\mathcal{P}$, *subtypeOf*) as the union of four posets with *OclVoid* as a lower bound. More formally, we have:

Definition 3.1 (subtypeOf). ($\mathcal{P}$, subtypeOf) = ($\mathcal{P}_c$, subtypeOf) $\cup$ ($\mathcal{P}_t$, subtypeOf) $\cup$ ($\mathcal{P}_p$, subtypeOf) $\cup$ ($\cup\mathcal{P}_m$, subtypeOf)

We consider that *OclAny* is the upper bound of $\mathcal{P}_p$ and $\cup\mathcal{P}_m$.

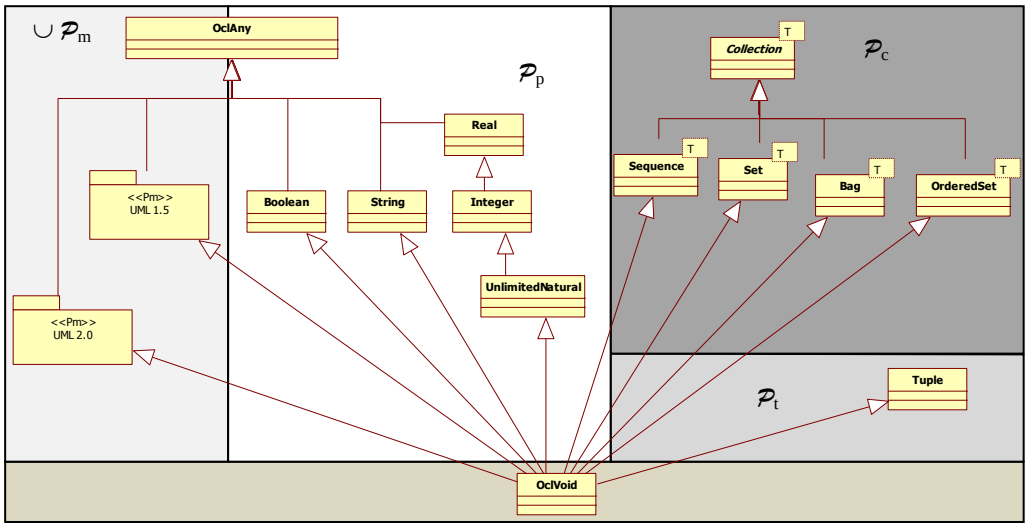

Fig. 3. Type hierarchy and the posets.

This choice conforms to the Amsterdam manifesto and the annex A of the standard that describes the basis of OCL [Cook et al. 2002]. This manifesto indicates that simple set inclusion semantics for subtype relationships would not be possible due to cyclic domain definitions if *OclAny* were the supertype of *Set(OclAny)*.

In OCL 2.2 standard, the type of collections conforms to *OclAny*. However, in the request for proposal emanated from the OMG Group concerned OCL 2.5 [OMG 2014a] indicates that:

> "proposals may revisit the OCL 2.2 change to conformance of Collection to OclAny, provided the change supports".

The two main arguments are:

(1) the interface of *OclAny* is too rich. Since a common supertype is required for the polymorphic behavior, the introduction of a new superclass with only two methods *isOclCollection()* and *isOclType()* [Brucker et al. 2013] was discussed.
(2) types of heterogeneous nested collections are stronger than *Collection(OclAny)*.

For a static typing system, the use of the type erasure technique [Allen et al. 2003] at run-time is a necessity. This implies that the type hierarchy must have an upper-bound named *OclAny*. In our approach we do not need to use this technique and so we conform to the OCL manifesto.

The abstract class *Collection* does not inherit from *OclAny*, but implements some of its features concerning management of types at run-time. In fact, we do not want to have collections mixing atomic, collection and tuple types. Having a common upper bound for the OCL type hierarchy facilitates the implementation of the static type evaluator with parameterized types [Allen et al. 2003]. In our case, the separation between the different posets provides a faster feedback to the designer of OCL rules. If the designer really wants mix tuples, atoms and collections into collections, he must use cast functions as for static typing policy.

3.2. The sCST operation

According to the ($\mathcal{P}$, *subtypeOf*) hierachies of Figure 3, two issues arise for the *sCST* operation. The first one is unifying impossibility for elements from different posets. The second one is that user metamodels ($\cup\mathcal{P}_m$) are not join-semilattices.

First, we propose our solution and then we define basic operations on types for the two type constructors in OCL: the collections and the tuples.
The first problem concerning the unification impossibility took us to introduce a precondition as follows:

(1) We make the hypothesis that the analysis stops when an *sCST* invocation fails. In this case, an instance of *OclInvalid* is returned. This is the consequence of the fact that the *sCST* operation is partially defined as previously mentioned.
(2) Considering $\mathcal{P}_{pm} = \mathcal{P}_p \cup (\cup\mathcal{P}_m)$, we define the operation
isUnifiable(t_1, t_2) = ($t_1 \in \mathcal{P}_{pm} \wedge t_2 \in \mathcal{P}_{pm}$) $\vee$ ($t_1 \in \mathcal{P}_c \wedge t_2 \in \mathcal{P}_c$) $\vee$ ($t_1 \in \mathcal{P}_t \wedge t_2 \in \mathcal{P}_t$) $\vee$ (t_1=*OclVoid*) $\vee$ (t_2=*OclVoid*).

The semantics of the *smallestCommonSuperType* operation is by definition the least upper bound of the ($\mathcal{P}$, *subtypeOf*) hierarchy. To satisfy the definition of the least upper bound this operation must respect the following three properties for *A* and *B* in $\mathcal{P}$:

Property 3.2 ensures that the *sCST* operation evaluates an upper bound.

Property 3.2. $\forall t \in sCST(A, B) / iSO(A, t) \wedge iSO(B, t)$

Property 3.3 ensures that the upper bound evaluated by the *sCST* is one of the least upper bound.

Property 3.3. $\forall t \in sCST(A,B), \forall t' \in \mathcal{P}, t \neq t' / iSO(t', t) \Rightarrow \neg(iSO(A, t') \wedge iSO(B, t'))$

Property 3.4 ensures the uniqueness of the least upper bound.

Property 3.4. $\forall t, t' \in \mathcal{P} / t \in sCST(A,B) \wedge t' \in sCST(A,B) \Rightarrow t=t'$

These three properties are valid on a partially ordered relationship represented by a join-semilattice [Levine 2011]. However, there is no indication in the standard allowing the assertion that metamodels are join-semilattices [Schürr 2002]. Without new hypothesis or specific treatment, the uniqueness of the least upper bound cannot be guaranteed. After a discussion about solutions proposed by A. Schürr in [Schürr 2002] to achieve the determinism of this operation, we give our solution and explain it along an example.

The first one is to impose that all metamodels are built as join-semilattices or that all metamodels are built using only single inheritance. These restrictions are quite obvious and difficult to accept for designers who define metamodels.

The second solution is to handle sets of OCL types and to define the appropriate type checking rules. Having a set of types instead of only one type requires the redefinition of the operation *sCST* and all the typing rules. A. Schürr promotes this solution because OCL rules are considered at M1 level which supports multiple classifiers for an instance [OMG 2003; OMG 2009].

The third solution is a preprocessing phase that checks the join-semilattice property for a given metamodel and adds the still needed intermediate classes without revealing their existence to the end user. The preprocessing of new classes may generate a combinatorial explosion of the metamodel size. Moreover, in a static typing policy, the programmer must choose one of the superclass that he wants use services. This is not the case in a dynamic typing policy where OCL rules are evaluated until a problem type appears in the evaluation of a sub-expression.

In this context, we have chosen to adapt the third solution to complete the metamodel on the fly during rule analysis. After the evaluation, all types created to complete the hierarchy are erased.

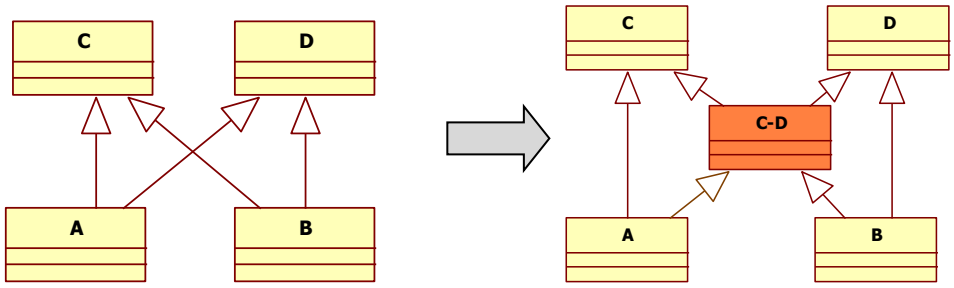

Fig. 4. A metamodel fragment and its completion.

The left part of Figure 4 shows a metamodel fragment comprising four classifiers *A*, *B*, *C* and *D*. This fragment is enriched by a volatile intermediate type. We annotate the diagram of Figure 4 with C-D to characterize the uniqueness of the least upper bound of *A* and *B*.

This situation occurs in the UML 1.5 metamodel [OMG 2003] for the classifiers *Class* (*A*), *Package* (*B*), *Namespace* (*C*) and *GeneralizableElement* (*D*) as shown in the Figure 5.

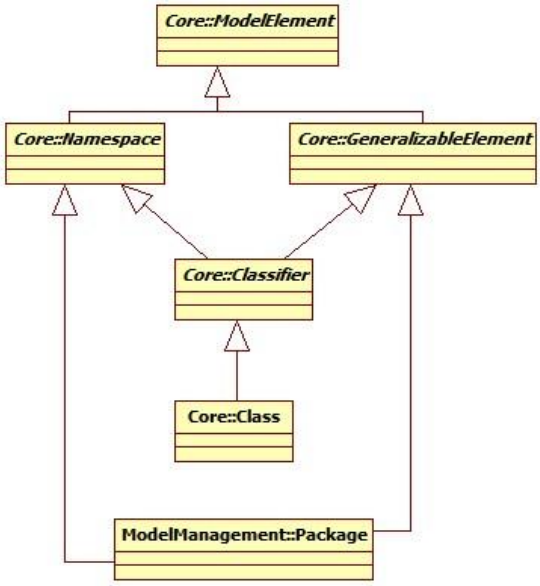

Fig. 5. Excerpt of the metamodel UML 1.5 for *Package* and *Class* hierarchy.

For example, the following OCL rule evaluates an expression mixing *Class* and *Package*:

OCL RULE 4. Exemple of rule on UML 1.5 for *Package* and *Class* hierarchy

```
context Foundation::Core::Class
inv :  ModelManagement::Package::allInstances->including(self)->size < 10
```

The evaluation of *ModelManagement::Package::allInstances->including(self)* requires to introduce an intermediate type for typing elements contained by the collection. Figure 5 shows that *Class* and *Package* have two smaller upper bounds: *GeneralizableElement* and *Namespace*. The name of the new generated type for the rule evaluation is the concatenation of the name of the two smaller upper bounds separated with a dash. Table III shows the typing interpretation of OCL rule 5.

Table III. Analysis step by step of OCL rule 5.

Expression	Type
ModelManagement::Package::allInstances	Set(ModelManagement::Package)
->including(self)	Set(Foundation::Core::GeneralizableElement-Foundation::Core::Namespace)
->size	Integer
< 10	Boolean

We interpret this rule in an optimistic way that permits to pursue the evaluation until a final result.

3.3. isSubtypeOf (iSO) for Collections and Tuples

To define this relationship between two collections, we represent an instantiation of *Collection* type by a couple $<t_c, t_e>$ where $t_c \in \mathcal{P}_c$ and $t_e \in \mathcal{P}$. Taking into account this representation, the *isSubtypeOf* relationship between two instances of *Collection* is defined by:

Definition 3.5. Given $c_1=<t_{c1}, t_{e1}>$,
$c_2=<t_{c2}, t_{e2}>$ $iSO(c_1, c_2) \Leftrightarrow iSO(t_{c1}, t_{c2}) \wedge iSO(t_{e1}, t_{e2})$

To define the binary relationship between two tuples, we represent an instance of tuple as a set of attributes. An attribute is a couple that consists of a name and a type of $\mathcal{P}$. The name of an attribute is unique to an instance. Taking into account the representation of an instance, the *isSubtypeOf* relationship between two instances is defined by:

Definition 3.6. Given $tu_1=\{att_{11}, .., att_{1i}, ..., att_{1n}\}$,
$tu_2=\{att_{21}, \ldots, att_{2j}, \ldots, attt_{2m}\}$
$iSO(tu_1, tu_2) \Leftrightarrow n \geq m \wedge \forall j \in [1, m]\ \exists i \in [1, n]$
$name_{1i}=name_{2j} \wedge iSO(type_{1i}, type_{2j})$

3.4. smallestCommonSuperType (sCST) for Collections and Tuples

Now we define the *sCST* operation for collections and tuples:

Definition 3.7. Given $c_1 = <t_{c1}, t_{e1}>$ and $c_2 = <t_{c2}, t_{e2}>$
$sCST(c_1, c_2) = <sCST(t_{c1}, t_{c2}), sCST(t_{e1}, t_{e2})>$

Concerning tuples, we define the *sCST* operation as:

Definition 3.8. Given $tu_1 = \{att_{11}, \ldots, att_{1n}\}$ and $tu_2 = \{att_{21}, ..., att_{2m}\}$,
$sCST(tu_1, tu_2) = \{att_{31}, \ldots, att_{3p}\} / \forall k \in [1, p], \exists i \in [1, n], \exists j \in [1, m]$
$name_{3k} = name_{1i} = name_{2j} \wedge type_{3k} = sCST(type_{1i}, type_{2j})$

The sCST operator is defined for primitive types, collections type constructors and types of metamodels by construction, as previously seen. Typing a collection involves both container and element types. Our definition of sCST on collections applies the operator sCST on each component type of the collection. For tuples, the typing involves the computation of the intersection of attributes with respect to their names. Our

definition of sCST on tuples applies the sCST operator on the types of unified attributes. These conditions on collections and tuples ensure the least upper bound properties.

3.5. OCL typing operators

The OCL language supports three operations for type checking or cast an OCL rule. Operations used to verify the type of a rule can be easily implemented using the operators defined below. Given a valid OCL expression *exp* and *type* the operation that computes the type of the expression:

(1) *oclIsTypeOf*(exp, c : $\mathcal{P}$) is equivalent to iSO(type(exp), c) and iSO(c, type(exp)).
(2) *oclIsKindOf*(exp, c : $\mathcal{P}$) is equivalent to iSO(type(exp), c).
(3) *oclAsType*(exp, c : $\mathcal{P}$) implies type(exp)=c.

We are skeptical about the standard specification for the use of the operator *oclAsType*. We found two different semantics: one for coercing types of object and one for selecting one of the properties defined in a superclass. The first use is formalized on [OMG 2010, p.13]:

> "Casting provides visibility, at parse time, of features not defined in the context of an expression's static type. It does not coerce objects to instances of another type, nor can it provide access to hidden or overridden features of a type".

In the second one it is mentioned on page 20 that:

> "whenever properties are redefined within a type, the property of the supertypes can be accessed using the *oclAsType* operation."

The first semantics of *oclAsType* is not necessary when using a dynamic typing policy. Each expression is correctly typed and the operations that can be applied are correctly selected. We have chosen to implement the *oclAsType* operation to select the desired operation into a type hierarchy as depicted in [Chiorean et al. 2008].

4. Dynamic typing of OCL rules

Our formalization requires the computation of missing types to verify expression types. Computing missing types consists of unifying types computed during rule analysis in order to ensure that all types are valid.

Type inference is defined in [Palsberg 1996] as

> "the computation of missing type annotations. It allows us to omit some or all of the type annotations in our programs, and it is successfully used in functional languages such as ML and Haskell".

In this context, type unification composes or combines multiple types in a consistent manner in order to resolve missing types.

Type checking is defined in [Steel and Jézéquel 2007], as a

> "process verifying that a program is consistent according to these rules, using the redundancy included by the programmer in the form of syntactic elements such as type annotations".

This section presents the formalization of the OCL type system using the natural deduction logic [OMG 2011b; Prawitz 2006]. We do not present all the typing rules, only the more significant: context, basic operators' dot and arrows, and main operators on collection and tuple types.

4.1. Context definition

An OCL rule is defined by a context named *Cl*. For us, the context is an element of $(\cup\mathcal{P}_m)$, an operation or an attribute name. We remind you that we verify rules at metamodel level.

For each new context, a special variable called *self* exists. This variable represents an instance of the context, and *Cl::allInstances()* provides a set of all the instances of *Cl*.

Table IV. Context typing rules.

$$\frac{Cl \in (\cup\mathcal{P}_m)}{\Gamma, Cl \vdash self : t \wedge iSO(t, Cl)} \quad ❶$$

$$\frac{Cl \in (\cup\mathcal{P}_m)}{\Gamma, Cl \vdash Cl::allInstances() : \langle Set, t\rangle \wedge iSO(t, Cl)} \quad ❷$$

Let us consider the environment Γ and *Cl* a metaclass part of this environment. Typing rule ❶ asserts that the type of *self* is subtype of the metaclass that define the context of the OCL rule.

Typing rule ❷ generalizes the typing rule ❶ when all instances of the metaclass are required.

4.2. Dot and arrow operators

OCL basic rules principally use two operators: dot and arrow. These operators apply the right operand (property of a metaclass or operation on collections) on the left rule, as in object-oriented programming paradigm.

If the dot operator is applied to a single element, the result is the application of the right operand on the element. If the left operand is a tuple, the right operand is either the name of a tuple attribute or an operation available on all types of the tuple attributes. If the left operand is a collection, a bag is built containing the result of the application of the right operand on each element of the collection.

The arrow operator is dedicated to collections, and the right operand is applied to the whole collection itself. For example the *size* operation evaluates the number of elements of a collection. The semantics of these operators implies that the right operand of both operations is an implicit parameter named the caller or the first parameter.

Table V. Dot and arrows typing rules.

$$\frac{\Gamma \vdash e : t \in \mathcal{P}_{pm} \quad \Gamma \vdash o : t_1 \rightarrow t_2 \wedge iSO(t, t_1)}{\Gamma \vdash e.o : t_2} \quad ❶$$

$$\frac{\Gamma \vdash e : t \in \mathcal{P}_t \text{ with } t=\{att_1, \ldots, att_n\} \wedge \exists i \in [1, n] / att_i = \{name_i, t_i\}}{\Gamma \vdash e.name_i : t_i} \quad ❷$$

$$\frac{\Gamma \vdash e : t \in \mathcal{P}_c \text{ with } t = <t_c, t_e> \qquad \Gamma \vdash o : t_1 \rightarrow t_2 \wedge iSO(t_e, t_1)}{\Gamma \vdash e.o : <Bag, t_3> \wedge iSO(t_3, t_2)} \quad ❸$$

$$\frac{\Gamma \vdash e : t \in \mathcal{P}_c \quad \Gamma \vdash o : t_1 \rightarrow t_2 \wedge iSO(t, t_1)}{\Gamma \vdash e->o : t_2} \quad ❹$$

$$\frac{\Gamma \vdash e : t \in \mathcal{P}_{pm} \cup \mathcal{P}_t \quad \Gamma \vdash o : <Bag, t_3> \rightarrow t_2 \wedge iSO(t, t_3)}{\Gamma \vdash e->o : t_2} \quad ❺$$

Typing rule ❶ deals with access on properties of metaclass and propagate the type of expression with the type of the acceded property.

In the case of tuples, typing rule ❷ propagates the type corresponding to the selected name.

Typing rule ❸ produces a bag of *t* as indicated in the OCL standard. Type *t* is the smallest common super type of all types of the application of the operation on the elements of the collection.

Typing rule ❹ is similar to typing rule ❶ for collections.

In order to simplify the writing of OCL rules it is possible to apply the arrow operator on elements from $\mathcal{P}_{pm} \cup \mathcal{P}_t$. The OCL standard considers that the operator is applied to a singleton encapsulating this element. The typing rule of this specific use is given by the rule ❺ of Table V.

4.3. Collection types

Collections support four sets of operations: constructors, observers, transformers, and iterators. The OCL specification does not clearly state how the operations on collections would be implemented. However, it seems that all the transformer operations applied to a collection generate a new collection as a result. Then, we consider that transformers are secondary generators.

Table VI. Constructors typing rules.

Γ |- *Collection*{ } : <t_c, OclVoid>	❶
Γ |- $e_1 : t_1$ Γ |- *Collection*{e_1} : <t_c, t_1>	❷
Γ |- $e_1 : t_1$ $\quad$ Γ |- $e_2 : t_2$ *isUnifiable* (t_1, t_2) Γ |- *Collection*{e_1, e_2} : <t_c, *sCST*(t_1, t_2)>	❸
Γ |- *Collection*{e_1,…, e_{n-1}} : <t_c, t_e> $\quad$ Γ |- $e_n : t_n$ *IsUnifiable* (t_e, t_n) Γ |- *Collection*{e_1,…, e_n} : <t_c, *sCST*(t_e, t_n)>	❹
Γ |- *Collection*{e_1,…, e_{n-1}} : <t_c, t_e> $\quad$ Γ |- $e_n : t_n$ *isUnifiable* (t_e, t_n) Γ |- *Collection*{e_1,…, e_{n-1}}->*including*(e_n) : <t_c, *sCST*(t_e, t_n)>	❺

The typing rules are based on the inductive construction of the collection itself. *Collection*{e_1,…, e_n} represents a type constructor of a collection with *Collection* ∈ {*Set*, *Bag*, *Sequence*, *OrderedSet*} and e_1,…, $e_n \in \mathcal{P}$. When the collection is empty we consider that the element's type of the collection is *OclVoid*, the lower bound of the hierarchy $\mathcal{P}$. Through generalization, we compute the element's type of a collection by unifying the current element's type of the collection and the type of the new included element as shown in typing rule ❺ in Table VI.

We propose now to present only significant examples in the following tables. For example, Table VII presents the observers' typing rules. We have elected to present here only observers supporting one parameter, *i.e.*, *includes*, *excludes*, *includesAll*, *excludesAll*, *indexOf* and *count*.

Table VII. Observers' typing rules.

Γ |- e : <t_c, t_e> $\quad$ Γ |-p : $t \in \mathcal{P}$ $\quad$ Γ |- oper : $\mathcal{P}_c \times \mathcal{P} \rightarrow \mathcal{P}_p$ *isUnifiable* (t_e, t) Γ |- e->oper(p) : $\mathcal{P}_p$	❶
Γ |- e : $t \in \mathcal{P}_{pm} \cup \mathcal{P}_t$ $\quad$ Γ |-p : $t_1 \in \mathcal{P}$ $\quad$ Γ |- oper : $\mathcal{P}_c \times \mathcal{P} \rightarrow \mathcal{P}_p$ *isUnifiable* (t, t_1) Γ |- e->oper(p) : $\mathcal{P}_p$	❷

For observer's rules, we consider the case when *e* is not a collection ❷. In this case, *e* is implicitly cast in a bag of *e* and typing rule ❷ of Table VII is a specific case of typing rule ❶ In the following, as for all collection operators, a rule would be provided when *e* is not a collection.

In secondary generator rules, other expressions with the product, union, or intersection operators exist. Indeed, all the combinations between two collections would be specified. However, these rules differ only in the result type product.

Table VIII. Product typing rule.

$$\frac{\Gamma \text{ |- } e : \langle t_c, t_e \rangle \qquad \Gamma \text{ |-} p : \langle t_{cp}, t_{ep} \rangle}{\Gamma \text{ |- } e\text{-> product}(p) : \langle \mathit{Set}, \{(\text{first}, t_e), (\text{second}, t_{ep})\} \rangle}$$

During runtime, each couple {(first, t_e), (second, t_{ep})} is dynamically typed.

Table IX. Union typing rules.

$$\frac{\Gamma \text{ |- } e : \langle \text{Bag}, t_e \rangle \quad \Gamma \text{ |- } p : \langle t_{cp}, t_{ep} \rangle \quad \mathit{isUnifiable}\ (t_{ep}, t_e)}{\Gamma \text{ |- } e\text{->union}(p) : \langle \text{Bag}, t \rangle \text{ with } \mathit{iSO}(t_e, t) \wedge \mathit{iSO}(t_{ep}, t)} \ \text{❶}$$

$$\frac{\Gamma \text{ |- } e : \langle \text{Set}, t_e \rangle \quad \Gamma \text{ |- } p : \langle \text{Set}, t_{ep} \rangle \quad \mathit{isUnifiable}\ (t_{ep}, t_e)}{\Gamma \text{ |- } e\text{->union}(p) : \langle \text{Set}, t \rangle \text{ with } \mathit{iSO}(t_e, t) \wedge \mathit{iSO}(t_{ep}, t)} \ \text{❷}$$

$$\frac{\Gamma \text{ |- } e : \langle \text{Set}, t_e \rangle \quad \Gamma \text{ |- } p_1 : \langle \text{Bag}, t_{ep} \rangle \quad \mathit{isUnifiable}\ (t_{ep}, t_e)}{\Gamma \text{ |- } e\text{->union}(p) : \langle \text{Bag}, t \rangle \text{ with } \mathit{iSO}(t_e, t) \wedge \mathit{iSO}(t_{ep}, t)} \ \text{❸}$$

Dynamic typing requires tagging (or annotation in [Cengarle and Knapp 2004]) each value with its type. For example a collection has a parametric type composed of the collection's type itself. In addition, each element of the collection is tagged with its own type that is a subtype of the collection's parameter.

The type computing for a collection is made easier by the fact that OCL is a side effect-free language. Each operation concerning collections generates a new collection. So the type of the collection, and in particular the parameter's type of the new collection, is dynamically built during the generation. If we consider the implementation of the *union* operation using the *iterate* operator, the calculus of the collection's type is dynamically built in parallel with the result of the *union*. For example, the *union* operation between two sets can be defined by:

Given e : <Set, t_e> and t : <Set, t_t>
e->iterate(it; res = t->iterate(it1; res1 = Set{} |
    res1->including(it1)) | res->including(it))

The *union* operation between a bag and a set can be defined by:

Given e : <Bag, t_e> and t : <Set, t_t>
e->iterate(it; res = t->iterate(it1; res1 = Bag{} |
    res1->including(it1)) | res->including(it))

With this implementation of the *union* operation, the type of the result is calculated using the typing rules defined in Table VI, Table VII and Table XI (which will be presented later).

We next give typing rules for the *intersection* operator and we show that typing rules are similar to the typing rules for the *union* operator.

Table X. Intersection typing rules.

$$\frac{\Gamma \text{ |- } e : \langle Bag, t_e\rangle \qquad \Gamma \text{ |- } p : \langle Bag, t_{ep}\rangle \qquad isUnifiable(t_{ep}, t_e)}{\Gamma \text{ |- } e\text{->intersection}(p) : \langle Bag, t\rangle \text{ with } iSO(t, t_e) \wedge iSO(t, t_{ep})} \;❶$$

$$\frac{\Gamma \text{ |- } e : \langle Set, t_e\rangle \qquad \Gamma \text{ |- } p : \langle t_{cp}, t_{ep}\rangle \qquad isUnifiable(t_{ep}, t_e)}{\Gamma \text{ |- } e\text{->intersection}(p) : \langle Set, t\rangle \text{ with } iSO(t, t_e) \wedge iSO(t, t_{ep})} \;❷$$

As for the *union* operation, it is possible to calculate in parallel the type and the result of an *intersection* considering its implementation using *iterate* operator. For example, the *intersection* between two bags can be defined by:

```
Given e : <Bag, te> and t : <Bag, tt>
e->intersection(t) =
    e->iterate(it : te; res : Bag(OclVoid) = Bag{} |
      if t->includes(it) then res->including(it) else res endif)
```

With this implementation of the *intersection* operation, the type of the result is calculated using the typing rules defined in Table VI, Table VII, Table XI and the typing rules concerning *if..then..else..endif* operation.

The last group of operations evaluates an OCL rule on all elements of a collection. An iterator has a special variable called an accumulator, enriched with the result of the application of the OCL rule on each element of the collection. The accumulator and specific variables of the iteration can be omitted when the clause *iterate* is used to implement other iterator operations. In this case, one implicit iterator variable is created and used to evaluate the result. However this rule can be generalized to several iteration variables. If t_{cur} is omitted the type of *cur* is *OclVoid*. If t_{acc} is omitted the type of *acc* is t_{val}. Other operators such as *select*, *reject*, *count*, *forall*, *closure* and so on can be expressed using the *iterate* operator [OMG 2010]. In the standard [OMG 2012, p.168] the *closure* operator is given with the use of the *iterate* operator. This implementation is very complex and requires the use of the *oclAsType* operator to cast different parts of the rule. Using our approach, this expression is easier to write and to understand.

Table XI. Iterator typing rules.

$$\frac{\Gamma \text{ |- } e : t = \langle t_c, t_e\rangle \qquad \Gamma \text{ |- } val : t_{val} \in \mathcal{P} \qquad \Gamma, cur : t_{cur} \in \mathcal{P}, acc : t_{acc} \in \mathcal{P} \text{ |- } exp : t_{exp} \in \mathcal{P} \qquad iSO(t_e, t_{cur}) \wedge iSO(t_{val}, t_{acc})}{\Gamma \text{ |- } e\text{->iterate}(cur : t_{cur}; acc : t_{acc} = val \mid exp) : t_{ite} \wedge iSO(t_{ite}, t_{val})} \;❶$$

$$\frac{\Gamma \text{ |- } e : t = \langle t_c, t_e\rangle \qquad \Gamma \text{ |- } val : t_{val} \in \mathcal{P} \qquad \Gamma, cur \in \mathcal{P}, acc \in \mathcal{P} \text{ |- } exp : t_{exp} \in \mathcal{P}}{\Gamma \text{ |- } e\text{->iterate}(cur; acc = val \mid exp) : t_{ite} \wedge iSO(t_{ite}, t_{val})} \;❷$$

Typing rule ❶ in Table XI considers the *iterate* operator where the iterator and the accumulator have their types explicitly provided. Typing rule ❷ considers the *iterate*

where the iterator and the accumulator have not had their types explicitly provided. In this case, the types are dynamically computed from the type of *e* and from the type of *val*.

5. The Neptune platform

The first version of the Neptune platform was created during the IST project named NEPTUNE (IST-1999-2001). The aims of the platform were the provision of different tools for checking models and generation of associated documentation. After the testing of this platform by different partners, some new improvements seem to be pertinent. The second version of the platform [Millan 2013] was developed by the MACAO team only, in the Computer Science Institute of Toulouse (IRIT). The purpose of this platform is to provide an OCL evaluator and an environment for the generation of metamodels, loading models and evaluation of OCL rules. The next part of this section provides an overview of the Neptune platform. Then, the implementation of the OCL type synthesizer based on the formalization presented in Sections 3 and 4 is discussed. Following this two implemented extensions are presented.

5.1. Some screenshots of the platform

As shown in Figure 6, the NEPTUNE GUI is composed of two areas. The first area on the left of the window contains three browsers that allow to navigate in metamodels, models, and methods of meta-classes. The second area contains internal frames for editing and executing OCL rules. Frame ❶ contains all the functionalities dedicated to create, save, and edit OCL rules. Frame ❷ allows to evaluate a rule and eventually save the result. Frame ❸ is the rule editor. This editor provides a syntactic coloration. OCL keywords are displayed in blue and bold italic style.

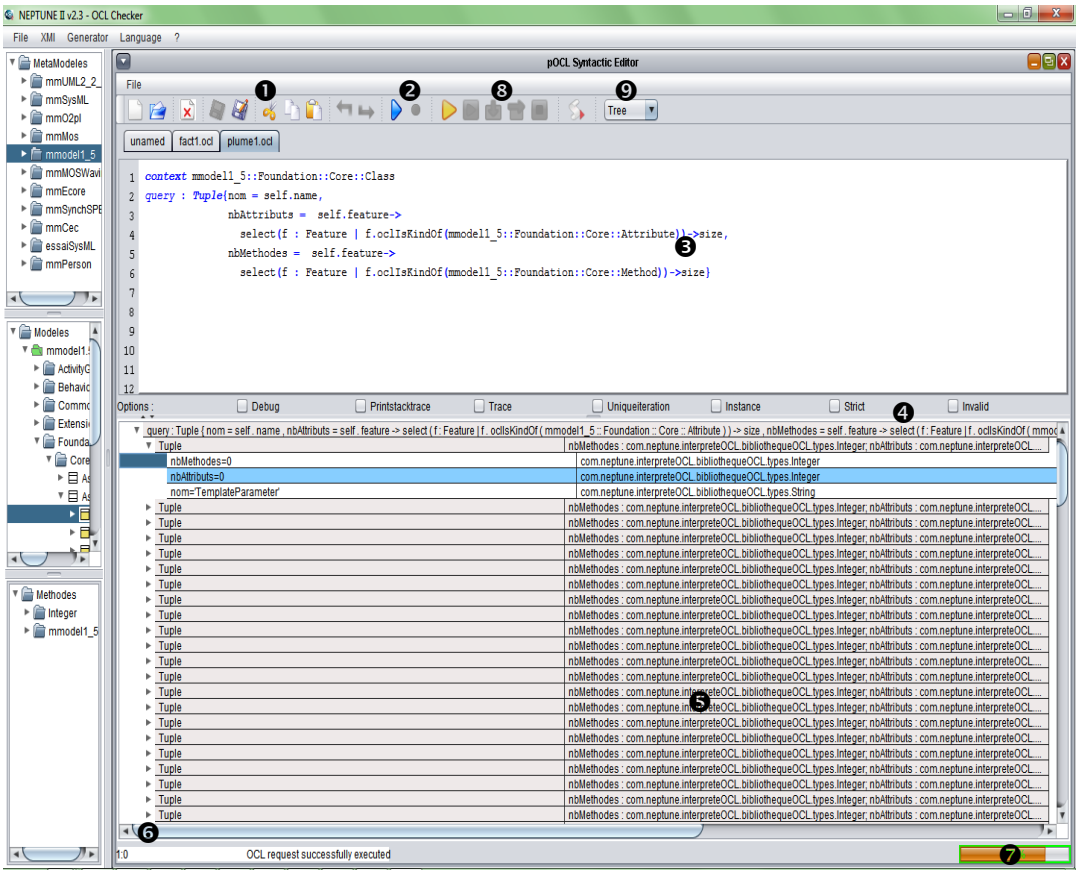

Fig. 6. NEPTUNE User Interface.

Frame ❹ permits to launch the OCL interpreter in different manners. In particular:

—UniqueIteration: this mode indicates that the rule is only executed for the first instance processed. This option can be used for example for rules that contain an *X.allInstances* feature.

—Instance: this mode indicates that the rule is only executed for the selected instance. The instance is selected in the model browser.

—Strict: this mode verifies that each expression is completely and correctly typed in the same manner as a static typing policy.

Frame ❺ displays the result of the evaluation of the rule. Each gray line displays the dynamic type of the rule processed on an instance. Frame ❻ permits to navigate into the log of the rule evaluation. Frame ❼ indicates the amount of memory available. Frame ❽ is dedicated to the debug mode. Frame ❾ is a drop-down list for selecting the view of the request evaluation.

Figure 7 illustrates the evaluation of a complex OCL rule in Frame ❸. This rule is extracted from the Bouhours' PhD. Thesis [Bouhours 2010] and aims to help programmers to isolate a model fragment substitutable with the Composite design pattern. Type and result of the evaluation is showed in Frame ❺.

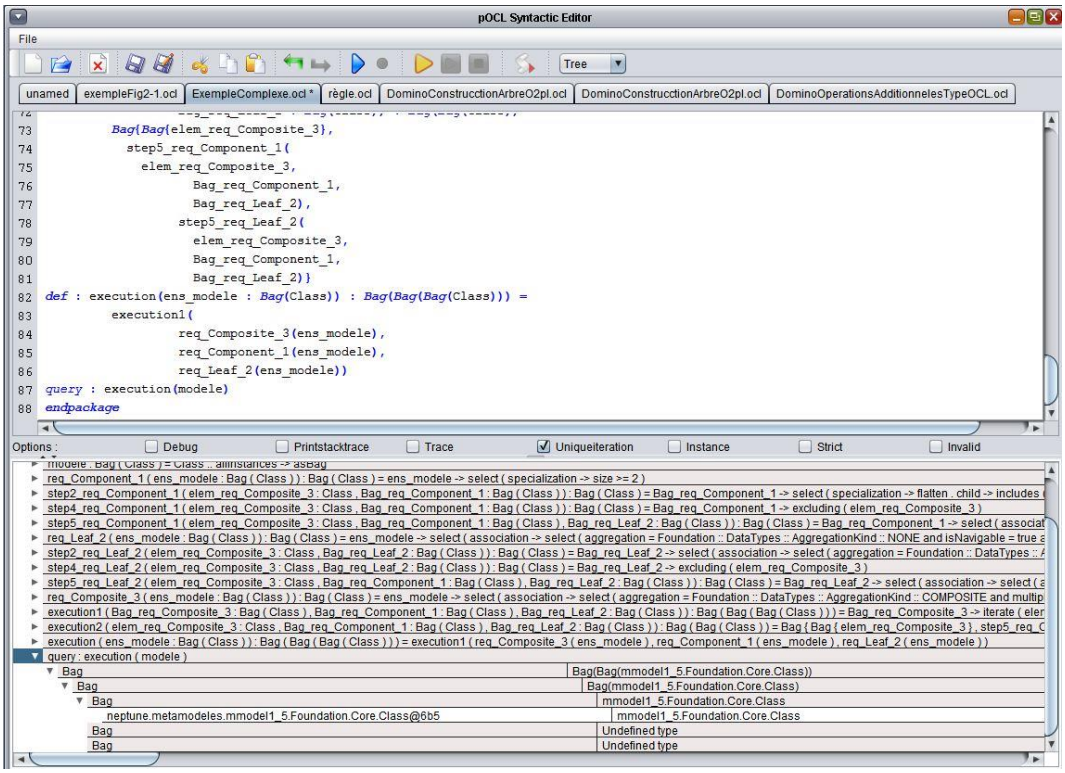

Fig. 7. Complex rule evaluation.

5.2. The OCL type analyzer

On the platform, all the metamodels are generated in the shape of Java class diagrams where a metaclass is transformed into a Java class and its interface. This implementation is that recommended by the gang of four in [Gamma et al. 1994], where they advocate separation of types depicted by Java interfaces and implementation depicted by Java classes. This implementation offers the advantage of providing a solution for implementing multiple inheritance that is intensively used in the definition of metamodels.

Towards the top, Figure 8 presents an extract in the UML metamodel 1.5 where a metaclass *Classifier* inherits from two metaclasses *Namespace* and *GeneralizableElement*. On the right part of the figure, we can show the corresponding implementation in the platform that uses the Java language. The class *Classifier* implements the *Classifier_int* interface and extends the class *NameSpace*. The inheritance relationship between *Classifier* and *GeneralizableElement* is transformed into a composition link. This strategy implies the implementation of all the operations provided by *GeneralizableElement_int* in *Classifier* by a simple delegation. The transformation of multiple inheritance into a single inheritance and one delegation scheme does not require the imposition of a superclass. The partial order induced by type hierarchies in metamodels is conserved by the inheritance relationship between interfaces. Then, interfaces are used to implement $\mathcal{P}_m$ posets.

The other posets $\mathcal{P}_p$, $\mathcal{P}_c$ and $\mathcal{P}_t$ depicting the OCL types are encoded statically in the type synthesizer and are loaded during the initialization of the platform. The poset $\mathcal{P}_m$ representing a metamodel is dynamically built during the rules analysis by accessing the Java interface hierarchy. Only the metaclasses required during the rule analysis are

computed by the OCL synthesizer. The main advantage of this implementation is that memory overloading is minimized. And, during the analysis of an OCL rule, types are computed as described in Section 4.

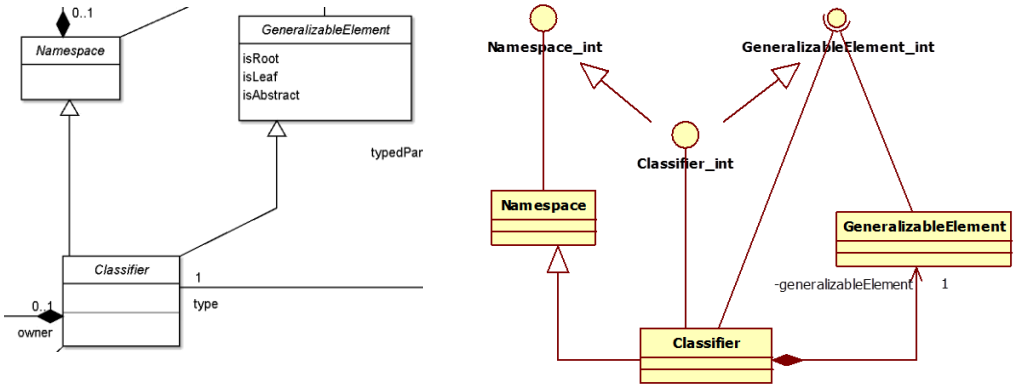

Fig. 8. Metamodel UML 1.5 excerpt and the counterpart of implementation.

6. Related work

Many studies have been carried out concerning the OCL language. Some of them concern the operational semantics of the OCL operators [Richters and Gogolla 2002], others specify the conformance relationship between OCL sub-expressions and elements of static UML diagrams [Clark 1999]. As we propose in our study, this work enriches the OCL type system with classes issuing from the static model that the author wants to constrain.

Based on the results of a long-term project to formalize the semantics of OCL 2.0 in Higher-Order Logic, a discussion about semantic issues of OCL is given in [Brucker et al. 2007]. OCL standard does not specify how correct typings are computed. Then Featherweight OCL [Brucker et al. 2014] has been defined in order to support strongly static typing by giving semantic properties of cast operators. Contrary to us, the authors assume that a pre-process eliminates all implicit conversions due to subtyping by introducing explicit casts, *e.g. oclAsType()*, for managing method overloading and construction of heterogenous collections.

Since the first version of OCL, several works have been carried out to formally set out the OCL type system. The main contributions on this topic are presented in [Schürr 2002; Cengarle and Knapp 2004; Kyas 2005]. All these contributions concern the formalization of a static type system for OCL.

A complete specification of the OCL type system using the static typing approach is proposed in [Cengarle and Knapp 2004]. Static typing approximates the type of a rule during compilation time. It ensures the correctness of the access to the features of the classes (attributes and methods). For example, it allows access to overridden attributes using the *oclAsType* operator and ensures the correctness of the attribute access during compile time and that the result is conformed to the expected type. Dynamic typing does not provide hypotheses for attributes and method accesses but it does calculate the precise type of an expression during analysis.

For static typing it is necessary to calculate the type before the rule analysis. For example, the static type of a rule that evaluates the intersection between two sets requires an operation calculating the greatest common supertype between two types. This

operation is proposed by Schürr in [Schürr 2002]. In addition, the type calculated statically may be more imprecise than the type calculated dynamically. For example, if we consider two sets *Set{1, 2, 3, 4.0}* and *Set{1, 2, 5.0}* their type is a set of reals; if we evaluate the expression *Set{1, 2, 3, 4.0}->intersection(Set{1, 2, 5.0})* the type statically calculated is *set(Real)* but its type dynamically calculated is *set(Integer)* because the result of this expression is *Set{1, 2}*. Our approach only requires the definition of the *sSCT* operator because the type of collection's operations is dynamically built.

We consider that the type hierarchy of OCL includes the inheritance relationship of the underlying metamodels. In this case, we consider that subtyping relation includes the inheritance relation. We also consider that a class defines a type. This choice conforms to the implementation of the OCL 2.0 standard where both OCL types and metaclasses are kinds of classifiers. The subtyping relation is sufficient because we have chosen to evaluate rules expressed at a metamodel level and in this context metaclasses are limited to attributes.

Note also that the inheritance relationship for OCL datatypes is debatable because in OCL 2.3 *UnlimitedNatural* is subtype of *Integer* but *UnlimitedNatural* does not inherit of *Integer*. Indeed, the "*" value is an unlimited integer and not an integer [OMG 2012, p.156]). Then the cast of this value to an integer raised an error.

The problem of the use of specific typing operators is described in [Kyas 2005]:

> "OCL constraints are fragile under the operations of refactoring of class diagrams, package inclusion and package merging. These operations, which often do not affect the semantic value of a constraint, can render constraints ill-typed. This essentially limits the use of OCL to a-posteriori specification of class diagrams."

In order to simplify an OCL expression and type handling, OCL 2.4 [OMG 2014b] introduces two new operators *selectByKind* and *selectByType.*

These operators factorize the expressions *c->select(e | e.oclIsKindOf(t)).oclAsType(t)* and *c->select(e | e.oclIsTypeOf(t)).oclAsType(t)*. This is a simplification of the writing of an OCL expression, but it is also a device for masking the overhead due to the use of static typing.

Dynamic typing decreases significantly the use of specific typing operators (*oclIsTypeOf*, *oclIsKindOf*, *selectByKind*, *selectByType* and *oclAsType*). These operators are highly coercive OCL rules in the fact that rules are very dependent on the underpinning class diagrams. Indeed, dynamic typing avoids to resort to these operators and at runtime delegates to the type synthesizer the computation of the correct type dynamically taking into account the underpinning class diagram.

7. Discussion

Our type system facilitates the extensibility of the OCL language by extending the operations available in the different OCL libraries dedicated to atomic types. Unfortunately, the poor support of parametric types and the existing ambiguities between sets and ordered sets [Büttner et al. 2010] prevent the definition of new operations on tuples and collections. It is therefore impossible to add an operation with a context such as *Set(Integer)* due to the incomplete support of parametric types.

Indeed, the OCL standard specifies that a context is given by a string called SimpleName, but the specification also mentions that

> "This production rule represents a single name. No special rules are applicable. The abstract syntax of a simpleNameCS String is undefined

> in UML 2.3, and so is undefined in OCL 2.3. The reason for this is internationalization" [OMG 2012].

In this case, considering that *Tuple(a : Integer; b : Real)* is a string, the type semantics about the tuple is lost, and *Tuple(a : Integer; b : Real)* is not a subtype of *Tuple(b : Real; a : Integer).* This is inconsistent with the conformance rules that are presented in the OCL specification.

For example, we are able to define the *initCapLetter* operation for *String* in order to convert the first letter of a word to uppercase. This implementation consists of using the *def* feature in the *String* OCL context. Handling OCL types are not clearly defined in the OCL specifications and this omission of information is a brake for the implementation as well as ensuring the quality of OCL tools. We consider that the OCL generic type system is second-class because generic types cannot appear in any context where conventional types appear [Allen et al. 2003].

The new types that complete the poset in order to have a lattice locally are dynamically added to the type system. Using the static typing approach, it is necessary that all new types be computed during the compile-time in order to be checked before runtime. Using dynamic typing, new types are computed only when these types are required for evaluating the OCL rule. Building these new types allows rules to be well-typed whereas these rules are considered as ill-typed in a static environment. This capability justifies the difference between type and class, and the use of the *isSubtypeOf* relationship instead of the inheritance relationship.

Some irrelevant implementations such as UML stereotypes weaken the OCL type system. As mentioned in [Belaunde 2010], adding stereotypes introduces pending problems because of their approximate semantics and their management on both M1 and M2 levels [OMG 2009]. Indeed there is no information that indicates if a stereotyped class defines a new type. The only semantics provided in [OMG 2009] stipulates that

> "a stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends".

Moreover, the semantics of the extension link does not indicate if this link is a kind of inheritance link or something else.

Due to these inaccuracies we have not yet implemented the stereotypes on the NEPTUNE platform. However, we are studying an implementation using profiles and stereotypes in specific contexts which do not consider extension links as inheritance links and stereotypes as new types.

Dynamic typing and type synthesis are time and memory techniques consuming. The implementation of the synthesizer requires careful attention to the use of resources and particularly when models handled are huge. To control the resources we use the Netbeans profiler that highlights the methods to be optimized. We have evaluated the time spent by the type synthesizer with three experimentations.

The first one, proposed by M. Gogolla [Gogolla et al. 2008], aims to compute the prime numbers up to 2048. This experimentation concerns only integers but makes a huge number of calculi that require type verification for each calculus. The total execution time is 195 seconds and overhead for handling types represents 10.53 seconds, *i.e.*, only 5.4 % of the total time.

The second experimentation concerns the checking of the conformance of a transformation of a UML 2.1 activity diagram to a DSL. The models are quite small (77

Kb) but include a large number of metaclasses and thus types to handle. The execution time to handle types represents 5.8 %: 2.5 seconds are necessary for the transformation process including 0.145 seconds for types.

The last experimentation concerns a structural pattern matching algorithm for pattern detection [Bouhours et al. 2010]. For this experiment, we have carried out a reverse engineering of the Java AWT and have found composite design patterns in this model. Here the model is quite large (40 Mb) and the OCL rules used to detect composites are more complex. The query needs 12.9 seconds for a first result with an overhead of 0.92 seconds for type checks; overhead is then about 7.2 % of the execution time. Caches keeping intermediary type computation allow the reduction of this time when another execution is done. In this case, this time represents at least 4.9 % of the execution time.

8. Conclusion and future work

In this paper, we presented a formalization of the OCL type system based on two operations: *isSubtypeOf* and *smallestCommonSuperType*. These operations handle the OCL type hierarchy, in particular the partially ordered set of types provided by the standard. Our type synthesis ensures a correct type for each sub-expression that is handled and removes cast operators for the end-user. Moreover, formalization the OCL type system helps to disambiguate the standard.

This formalization was implemented by a type synthesizer on top of the NEPTUNE platform. This gave us the opportunity to evaluate the cost of the type synthesis in terms of evaluation duration and memory consumption. Using caches and optimizations, we reduced the type synthesis duration to 5 % of the total execution time. This overhead did not significantly increase the analysis process itself.

Currently we are focusing our work on benchmarking our platform to have a qualitative overview of its capability and formally validate the type synthesizer. As a starting point for this benchmark we consider M. Gogolla's works [Gogolla et al. 2008] which provide a set of rules for validating OCL evaluators. However, his work remains incomplete and requires, amongst other things, the addition of rules to completely validate the type system we are developing. For example, OCL rules of Section 2.4 could be added to validate the type system when types are calculated both statically or dynamically.

We also want to implement the *OclModel* type that takes into account the work realized by Steel in his thesis [Steel and Jézéquel 2007]. Implementation of a model type operator could contribute to the definition of specific operations for model transformation or composition of OCL rules shared by different models conform to different metamodels. Support of strong typing offers a frame for the definition of new types and associated operations as benefits.

Finally, our implementation of a dynamic type system for the OCL language is the first brick for implementation of an IDE for OCL that allows the loading of models, the writing of OCL rules and the execution of these rules as mentioned in [Chimiak-Opoka et al. 2010].

Concerning the OCL language itself, a new major version is planned [Willink 2014] for 2016, this version has for purpose to improve the language, type constructions and its libraries. Unfortunately, no significant advances were proposed concerning the introduction of dynamic typing as in others functional languages.

REFERENCES

E. Allen, Jonathan Bannet, and Robert Cartwright. 2003. A First-class Approach to Genericity. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*. OOPSLA '03. New York, NY, USA: ACM, 96–114. DOI:http://dx.doi.org/10.1145/949305.949316

B. Baudry et al. 2010. Trust in MDE Components: the DOMINO Experiment. In *SD4RCES workshop in conjunction with SAFECOMP 2010*. Vienna.

M. Belaunde. 2010. Evolution of the OCL OMG Specification.

C. Bouhours, H. Leblanc, C. Percebois, and T. Millan. 2010. Detection of Generic Micro-architectures on Models. In IARIA, ed. 34–41.

C. Bouhours, "Detection, Explanations, and Refactoring of design defects: the spoiled patterns," phd, Université de Toulouse, Université Toulouse III - Paul Sabatier, 2010.

A.D. Brucker et al. 2013. Report on the Aachen OCL Meeting. 1092 (2013), 103–111.

A.D. Brucker, Jürgen Doser, and Burkhart Wolff. 2007. Semantic issues of OCL: Past, present, and future. (2007).

A.D. Brucker, Frédéric Tuong, and Burkhart Wolff. 2014. Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5. *Arch. Form. Proofs* 2014 (2014).

H. Bruneliere, Ronan Fortin, and Thierry Fortin. 2015. ATL/User Guide - Introduction. (February 2015). Retrieved June 17, 2016 from https://wiki.eclipse.org/ATL/User_Guide_-_Introduction

F. Büttner, M. Gogolla, L. Hamann, and M. Kuhlmann. 2010. On Better Understanding OCL Collections or an OCL Ordered Set is not an OCL Set.

M.V. Cengarle and A. Knapp. 2004. OCL 1.4/5 vs. 2.0 Expressions formal semantic and expressiveness. *Softw. Syst. Model.* 3, 1 (2004), 9–30.

J. Dobroslawa Chimiak-Opoka et al. 2010. OCL Tools Report based on the IDE4OCL Feature Model. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 44 (2010).

D. Chiorean, V. Petrascu, and D. Petrascu. 2008. How My Favorite Tool Supporting OCL Must Look Like.

T. Clark. 1999. Typechecking UML static models. In LNCS, ed. 503–517.

S. Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. 2002. The Amsterdam manifesto on OCL. *Lect. Notes Comput. Sci.* 2263, TUM-I9925 (2002), 115–149.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional.

M. Garcia and A. Jibran Shidqie. 2007. *OCL Compiler for EMF*,

M. Gogolla, M. Kuhlmann, and F. Büttner. 2008. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In LNCS, ed. 446–459.

M. Gogolla, Fabian Büttner, and Mark Richters. 2007. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.* 69 (2007), 27–34.

A.K. Jones and B.H. Liskov. 1978. A language extension for expressing constraints on data access. *Commun. ACM* 21, 5 (May 1978), 358–367.

F. Jouault and I. Kurtev. 2006. On the architectural alignment of ATL and QVT. In Communication of ACM, ed. *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*. 1188–1195.

M. Kyas. 2005. An extended type system for OCL supporting templates and transformations. In LNCS, ed. 83–98.

L. Levine. 2011. Algebraic Combinatorics. 18.312. (2011).
M. Mauny. 1995. *Functional programming using Caml Light*, INRIA.
T. Millan. 2013. The Neptune platform. (February 2013).
T. Millan, L. Sabatier, T.T. Le Thi, P. Bazex, and C. Percebois. 2009. An OCL extension for checking and transforming UML Models. In WSEAS Press, 144–150.
OMG. 2014a. Object Constraint Language (OCL) 2.5 - Request For Proposal. (March 2014).
OMG. 2010. *Object Constraint Language, OMG Available Specification, Version 2.2 - formal/2010-02-02*,
OMG. 2012. *Object Constraint Language, OMG Available Specification, Version 2.3.1 - formal/2012-05-09*,
OMG. 2014b. *Object Constraint Language, OMG Available Specification, Version 2.4 with CB - formal/2014-02-02*,
OMG. 2011a. *OMG Meta Object Facility (MOF) Core Specification - OMG Available Specification Version 2.4.1 - formal/2011-08-07*,
OMG. 2009. *OMG Unified Modeling Language (OMG UML), Superstructure. OMG Available Specification Version 2.2 - formal/2009-02-03*,
OMG. 2011b. *OMG Unified Modeling Language (OMG UML), Superstructure OMG Specification Version 2.4 - formal/2010-05-06*,
OMG. 2003. *OMG Unified Modeling Language (OMG UML), UML 1.5. OMG Available Specification Version 1.5 - formal/03-03-01*,
J. Palsberg. 1996. Type Inference for Objects. *ACM Comput. Surv.* 28, 2 (1996), 358–359.
B.C. Pierce. 2002. *Types and Programming Languages*, MIT Press.
D. Prawitz. 2006. *Natural deduction, a proof-theoretical study*, Dover Publications.
M. Richters and M. Gogolla. 2002. OCL: Syntax, Semantics and Tools. *Lect. Notes Comput. Sci.* 2263 (2002), 42–68.
A. Schürr. 2002. A new type checking approach for OCL Version 2.0. In LNCS, ed. 21–41.
J. Steel and J.M. Jézéquel. 2007. On model typing. *J. Softw. Syst. Model.* 6, 4 (2007), 401–413.
TOPCASED. 2010. *Règles OCL de modélisation*, TOPCASED.
TOPCASED. 2005. *SAM editor specification*, TOPCASED.
L. Tratt. 2009. Chapter 5 Dynamically Typed Languages. In BT-Advances in Computers, ed. Elsevier, 149–184.
C. Wilke, M. Thiele, and B. Freitag. 2011. *Dresden OCL manual for installation, use and development*,
E. Willink. 2006. Missing specification of UnlimitedNatural. (May 2006).
E. Willink. 2014. OCL 2.5 Plans. (September 2014). http://www.software.imdea.org/OCL2014/slides/OCL25Plans