Electronic Notes in Theoretical Computer Science 44 No. 4 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 14 pages

Adding Graph Transformation Concepts to UML's Constraint Language OCL

Andy Schürr [1]

Institute for Software Technology
University of the German Armed Forces, Munich
Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany

Abstract

The Object Constraint Language OCL is an integral part of UML, the Unified Modeling Language standard. It has been added to UML as a logic-based sublanguage for the definition of class invariants and pre-/postconditions of operations. OCL is rather similar to a subset of the graph transformation language PROGRES, the so-called path expressions. These path expressions are used for similar purposes as OCL. In contrast to OCL, path expressions support functional abstraction and offer additional operators for conditional iteration and transitive closure. Furthermore, PROGRES possesses a visual query sublanguage and is equipped with a precise semantics definition. Based on our experiences with the development of PROGRES a number of modifications and extensions of OCL are suggested as recommendations for its forthcoming version 2.0.

1 Introduction

The *object constraint language OCL* is an integral part of the Unified Modeling Language Standard UML [8] for the logic-based definition of class invariants or pre- and postconditions of operations [19]. In its current form OCL suffers from the same problems as many parts of the UML standard: it neither possesses a precise static semantics nor a precise dynamic semantics definition [14]. These are the reasons, why groups of researchers are now active to refine and redesign parts of OCL in order to influence the UML 2.0 definition process [3,13].

It is the purpose of a recently started research project to apply our experiences with the development of the *graph transformation language PROGRES* to OCL. PROGRES is a visual, executable specification language that combines (1) a subset of UML class diagrams for the definition of graph schemata

[1] Email: Andy.Schuerr@unibw-muenchen.de

with (2) graph transformation rules for the definition of object structure manipulations, and (3) OCL-like path expressions for the definition of integrity constraints, pre-/postconditions, and graph queries [17].

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

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

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

(i) Our path expressions have a well defined set of type checking rules expressed as predicate logic formulas.

(ii) Furthermore, PROGRES has a precisely defined semantics based on nonmonotonic reasoning and fixpoint theory.

(iii) Its dynamic semantics definition distinguishes between terminating computations that return the undefined result `nil` and nonterminating computations with unknown results.

(iv) Partially defined Boolean expressions, which caused the introduction of a three-valued logic in OCL, are disallowed; this is due to our experience that a three-valued logic often leads to rather unexpected results.

(v) Functional abstraction of ordinary expressions and path expressions (derived binary relationships on graphs) is supported.

(vi) Operators for the definition of default values (for partially defined subexpressions), building the transitive closure, conditional iteration etc. are available, which could be added to OCL.

(vii) The OCL-like textual path expression sublanguage of PROGRES is complemented by a graphical sublanguage which is closely related to the structural part of UML collaboration diagrams.

(viii) Attributed associations, n-ary associations, bags and sequences are not supported, despite of the fact that PROGRES users often complain the lack of these OCL features.

(ix) An integrated programming environment is available, comprising a syntax-directed editor, a parser, an incremental type checker, an interpreter, and a compiler to C.

As a consequence it was quite tempting to start a research project which refines the wide-spread OCL standard based on our experiences with the more or less unknown specification language PROGRES. First results concerning

the definition of a *more elaborate type system* for OCL, which address some problems not solved in [2]—are presented in [16] and will not be repeated here. It is the purpose of this paper to address items (vi) and (vii) above.

The following Section 2 starts with the introduction of a running example adopted from [11]. It uses a radically simplified meta model of UML class diagrams and collaboration diagrams (snapshots) to demonstrate typical difficulties of OCL with the definition of appropriate invariants. Section 3 shows how some *additional operators* taken from PROGRES simplify the construction of the OCL constraints of Section 2 considerably. Furthermore, this section introduces the concept of functional abstraction for OCL expressions and discusses some problems concerning the semantics of recursively defined OCL expressions.

Section 4 addresses the second main topic of this paper, the development of a *visual constraint definition sublanguage*. It starts with a short survey of OCL-related visual constraint definition languages published in [1] and in [11]. Based on the graph transformation related work presented in [1] and the related graphical constraint definition sublanguage of PROGRES we discuss the collaboration-diagram-based visual definition of constraints as well as different possibilities to define the semantics of graphical negation constructs. Finally, Section 5 summarizes the presented proposal and sketches future activities in this area.

2 The Running Example

The developer of one of the first proposals for mixing textual OCL constraints with a graphical constraint definition sublanguage uses in [11] a simplified fragment of the UML meta model as his running example. The selected fragment of a part of the UML class diagram meta model and a related part of its collaboration diagram model fulfills the following requirements:

- All readers familiar with UML or one of its predecessors have no problems to understand the depicted example and its related constraints.
- There are no doubts that the outlined problems with the OCL definitions of the needed constraints are of practical relevance.
- In its simplified form the example is still small enough to be used in a short paper like this one.
- The selected example highlights quite a number of problems with the currently valid definition of OCL.

We adopted this example with some minor modification of the accompanying meta class diagram shown in Figure 1. First of all, pairs of association-end-names were replaced by a single association (end) name. This is due to the fact that all needed OCL expressions navigate only in one direction along the regarded associations. Furthermore, we reintroduced the meta classes `(Model)Element` and `Instance` in order to reduce the number of needed dif-

ferent meta associations. Next, we decided that binary `Links` are not instances of `Roles`, but instances of binary `Assoc(iations)`. Finally, we added the import relationship `import` between packages as well as the meta associations `conformsTo` and `visible` as examples for the recursive definition of derived relationships.

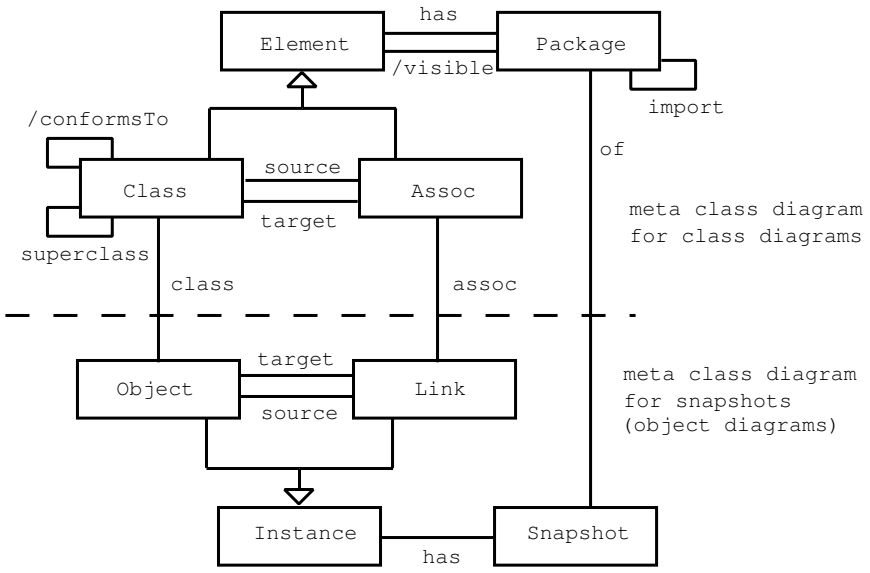

Fig. 1. The running example, a simplified UML meta model.

The following Figure 2 presents an example for an instance of the meta class diagram of Figure 1. Its upper part contains a package `Q` which introduces three classes (`A`, `B`, and `C`) and two associations (`a` and `b`) between them. The triangle "adornments" of the assocations indicate the fact that instances (links) of assocation `a` have instances (objects) of class `C` as source and instances (objects) of class `A` as target, whereas instances of assocation `b` have instances of class `C` as source and instances of class `B` as target. The class diagram second's package `P` imports all classes and assocations from package `Q`.[2] It uses the imported class `A` to derive the two new subclasses `D` and `B`. As a consequence, the new class `P::B` hides the imported class `Q::B` for package clients which use the simple class name `B` instead of the qualified class name `<package>::B` to reference a needed class.

The bottom part of Figure 2 presents one example of a snapshot (collaboration diagram) which is consistent with the class diagram `of` the related package `P` (including the visible part of the class diagram inside package `Q`). It uses a link (instance) of association `a` to connect an instance `o1` of class `C` to an instance `o2` of class `D`, which is a subclass of the required target class `A`.

The usually required constraints which guarantee *the consistency of a snapshot* with the class diagram of a selected package (`P` in this case) are:

[2] We assume that all elements shown in Figure 2 have visibility `public`.

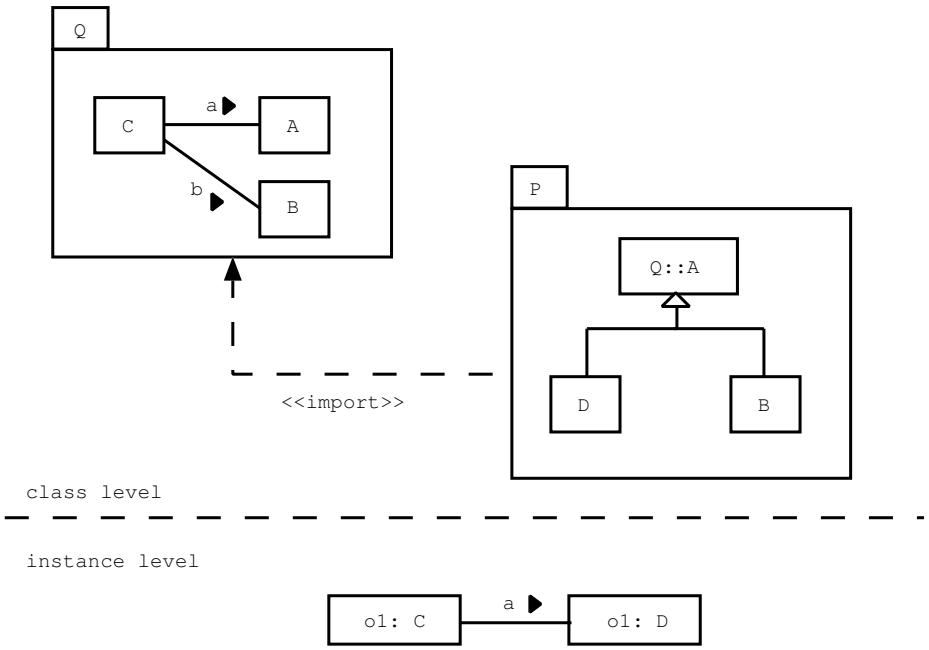

Fig. 2. Examples of a UML class diagram and a consistent snapshot.

(i) Any object of the snapshot is an instance of a class which is a visible component of the regarded package `P`.

(ii) Any link of the snapshot is an instance of an association which is a visible component of the regarded package `P`.

(iii) Any (binary) link of the snapshot has a source (target) object which is an instance of a class which conforms to the source (target) class of its association.

(iv) A class or association is visible for a package `P` if it is a component of `P` or a component of another package `Q` directly or indirectly imported by package `P` via `import` dependencies.

(v) A class `D` conforms to another class `A` if `D = A` or if there exists another class `B` such that `D` conforms to `B` and `B` is a direct subclass of `A`.

A precise definition of these restrictions using OCL version 1.3 as defined is not possible for the following reasons: The currently valid OCL standard neither offers functional abstraction nor a transitive closure operator needed for the definition of the two derived relationships `visible` and `conformsTo`. It is, therefore, current practice to assume the existence of functional abstraction and to use recursively defined functions (cf. related assertions of the UML standard meta model in [8]).

Using this extension of OCL it is possible to define the above listed constraints. Figure 3 starts with the OCL expressions for the main constraints (i), (ii), and (iii). The first OCL expression starts e.g. at an instance of class `Snapshot`, navigates via an `of` link to all `visible Class` instances of

```
(1) context Snapshot inv:
    self.of.visible.oclAsType(Class)->
         includesAll(self.has.oclAsType(Object).class)

(2) context Snapshot inv:
    self.of.visible.oclAsType(Assoc)->
         includesAll(self.has.oclAsType(Link).assoc)

(3) context Snapshot inv:
    self.has.oclAsType(Link)->forAll( l |
        l.source.class.conformsTo->includes(l.assoc.source)
    and
        l.target.class.conformsTo->includes(l.assoc.class) )

(4) context Package inv:
    self.visible = P.has->union(P.import.visible)->asSet

(4') context Package inv:
    self.visible(N) = if self.has->select(Name = N)->notEmpty then
                        self.has->select(Name = N)
                      else
                        self.import.visible(N)
                      endif

(5) context Class inv:
    self.conformsTo =
         self.superclass.conformsTo->including(self)->asSet
```

Fig. 3. OCL definitions of snapshot constraints.

the regarded package. It requires then that the result set of objects includes all `Class` instances of the regarded snapshot's `Objects`. The second OCL expression has a similar form. It requires that the associations of all links of the regarded `Snapshot` instance `self` are visible for its related package `self.of`.

The third OCL expression of Figure 3 deals with requirement (iii) above. It retrieves all `Link` instances of the regarded snapshot and requires for all elements `l` of this set that the `class` of the `source` (`target`) of its association `assoc` is an element of the set of all classes which are in `conformsTo` relation to the `class` of the `source` (`target`) of `l`.

The following OCL expression with label (iv) introduces the simple version of the derived relationship `visible` used above. The semantics of its recursive definition—computing the transitive closure of the association `import` followed by the traversal of the assocation `has`—has a more or less precisely defined semantics as long as there are no `import` loops in the regarded class diagram. It is an open question how a precise OCL semantics definition would handle *cyclic import relationships*. Following the lines of some deductive database system programming languages we might assume the so-called "closed world assumption" and use a bottom-up evaluation process for the computation of derived relationships. In this case, any component of any package on a cycle would be visible for all packages on the cycle. On the other hand, following a "naive" operational semantics definition approach the evaluation of the derived relationship (function) `visible` would never terminate applied

to a package on an `import` dependency cycle. We will see in the following Section 4 how PROGRES solves and how OCL version 2.0 might solve this problem by introducing a new transitive closure operator.

The following expression (4') introduces a slightly more complex version of the derived relationship (function) `visible`. Applied to a given package with a certain name `N` as parameter it finds a visible component with name `N`. Applied e.g. to package `P` of Figure 2 with parameter "`B`" it returns the class with name `B` of package `P` and not the class `B` of package `Q`. This expression gives us later on in Section 4 the motivation for the introduction of yet another OCL operator, which overcomes the duplication of the subexpression `self.has->select(Name = N)` shown in Figure 3.

The last expression with label (v) of Figure 3 has been added for reasons of completeness. It introduces the still missing definition of the derived relationship `conformsTo`. Again its semantics is undefined if we regard class diagrams with cyclic `superclass` relationships. Please note that a derived relationship of this kind would be necessary to define an OCL constraint which forbids cycles in the class hierarchy. As a consequence, a derived relationship like `conformsTo` is needed for excluding situations for which the semantics of `conformsTo` is not precisely defined.

3 New OCL Operators Adopted from PROGRES

This section proposes a number of extensions for OCL version 1.3/1.4 which solve some difficulties with the definition of derived relationships reported in Section 2. First of all a concept for *functional abstraction* is added to OCL. This allows one to encapsulate and reuse a navigational expression `E` which starts at an object of class `SC` and returns either a single well-defined object of class `TC` or a maybe undefined object of this class or a collection (set, bag, sequence) of objects of this class[3]. Such a functional abstraction may have additional parameters as shown in Figure 4.

The functions with label (4), (4'), and (5) of Figure 4 correspond to the OCL expressions with the same labels of Figure 3. The first function solves the previously mentioned termination problem with cyclic `import` relationships by using the new *transitive closure operator* `*`. The implementation of this operator keeps track of all already visited objects and avoids thereby any termination problems. Its translation into standard OCL or, more precisely, into OCL version 1.3 plus the vaguely defined functional abstraction used in many documents is presented in Figure 5. Please note that the recently added `let`-construct is probably used with the intended concrete syntax instead of the syntax defined in [8] which omits the expression after the keyword `in`. Furthermore, it is probably used with the intended semantics in mind, i.e. for

[3] The result parameter types are denoted as `TC` for a single result object, `TC?` for a maybe undefined result object, and `set/bag/sequence(TC)` for sets, bags, and sequences of objects of class `TC`.

```
(4)  function visible: Package -> set(Element) =
       self.import*.has->asSet
     end

(4') function visible(N: string): Package -> Element? =
       self.try
              has->select(Name = N)
            else
              import.visible(N)
            endtry -> unique
     end

(4") function visible(N: string): Package -> Element? =
       self.loop
              import
            exit with
              has->select(Name = N)
            endloop -> unique
     end

(5)  function conformsTo: Class -> set(Class) =
       self.superclass*->asSet
     end
```

Fig. 4. Modified OCL expressions with new operators.

assigning names to subexpressions and not for the definition of parametrized functions. From our point of view different syntactic constructs should be used for these two different purposes.

The first function defined in Figure 5, the recursively defined `pClosure` operator, uses an additional set-valued argument for keeping track of all already visited objects and. It excludes these objects from the set `S`, the input for its recursive call. The second function with label (4') demonstrates the usage of the two new operators `try` and `unique`. The evaluation of the `try`-operator starts with the evaluation of its first branch `has->select(Name = N)`. This branch tries to return an element with the required name which belongs to the current package. If and only if the evaluation of the first branch fails (returns the empty set) then the second branch `import.visible(N)` is evaluated which determines all imported packages of the current package and asks these packages for a visible element with the required name. Again we have the problem with termination if we assume a "naive" left-to-right and depth-first-evaluation of recursive function definitions. Therefore, we need a third new operator called `loop` which supports conditional iteration (with cycle check). The function declaration with label (4") uses this operator. It applies the subexpression `import` to all objects of a current set of objects (iteratively) for which the evaluation of the subexpression `has->select(Name = N)` fails.

For further details concerning the semantics of the `try`- and `loop`-operator the reader is referred to Figure 5. Its equation (2) translates the `try`-operator into standard OCL, whereas its equations (3a), (3b), and (3c) introduce different versions of the `loop`-operator. The first one iterates the application of its subexpression `P1` until the subexpression `P2` returns a nonempty set. This set

```
(1) X.p* = X.pClosure(X)
    X.pClosure(Y) = let S:... = X.p->excludeAll(Y) in
                      S->collect(pClosure(S->union(Y)))->asSet
                    endlet

(2) X.try P1 else P2 endtry =
      if X.P1->notempty then X.P1 else X.P2 endif

(3a) X.loop P1 until P2 endloop =
       X.( select(P2->isEmpty)->collect(P1) )* ->collect(P2)
     if P2 is an OCL expression which returns a set of objects

(3b) X.loop P1 until P2 endloop =
       X.( select(not P2)->collect(P1) )* ->select(P2)
     if P2 is a Boolean OCL expression

(3c) X.loop P endloop =
       X.P*->select(P->isEmpty)
```

Fig. 5. Translation of new operators in standard OCL.

is the result set of the whole expression. The second version permits a Boolean subexpression `P2`. It iterates the application of the subexpression `P1` for all objects which do not fulfill condition `P2`. It returns all visited objects which fulfill condition `P2`, including the start object in the general case. The third version of the `loop`-operator is just a short-hand for the often needed case that `P1` and `P2` are the same expressions. The expression `loop P endloop` returns all objects visited by the transitive closure of `P` for which the application of `P` returns the empty set. Applied to an object on a `P`-cycle `loop P endloop` computes therefore the empty set of objects.

Both function (4') and (4") of Figure 4 use a new `unique` operator to transform a singleton set back into a single element (from the type-checker's point of view). The `unique` operator returns an undefined result for nonsingleton sets, it returns the single element of the regarded set otherwise. As a consequence both versions of the function `visible` have the result type `Element?` for a maybe undefined result object of class `Element`.

4 A Visual Constraint Definition Sublanguage

Using the now precisely defined derived relationships `visible` and `conformsTo` of the previous section we will discuss how to make the constraints of Figure 3 more readable. It is often said that visual expressions are more readable than their textual counterparts. Therefore, UML is a collection of mainly visual modeling sublanguages with the exception of OCL, a purely textual sublanguage. As a consequence some proposals have already been made to add some visual constraint definition constructs to OCL, too. The first proposal of this kind we aware of suggests to use a variant of Euler circles [6] or Venn diagrams [18], which are called *spider diagrams* [11]. The following Figure 6 shows a definition of the constraint (i) of Figure 3 as a spider diagram. It requires that

any `Snapshot` is related (via an `of`-link) to a single `Package` [4] such that for all `Objects` of the regarded `Snapshot` the following conditions holds [5]: the set of `Class` instances of these objects is a subset of the set of `Class` instances belonging to the related `Package`.

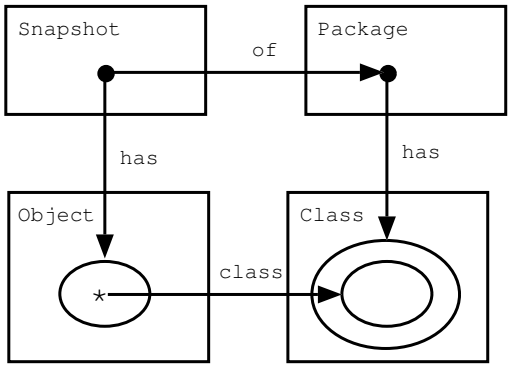

Fig. 6. Visual constraint definition with spider diagrams.

It is as usual a matter of taste and education whether the spider diagram of Figure 6 is more readable than the textual OCL version of Figure 3 or the other way round. But it is clear that spiders introduce yet another class of diagrams into UML. Therefore, another proposal has been made in [1] to use a variant of collaboration diagrams for the same purpose. These collaboration diagrams are interpreted as graph patterns of the so-called *algebraic graph transformation approach* as implemented by the graph transformation system AGG [5].

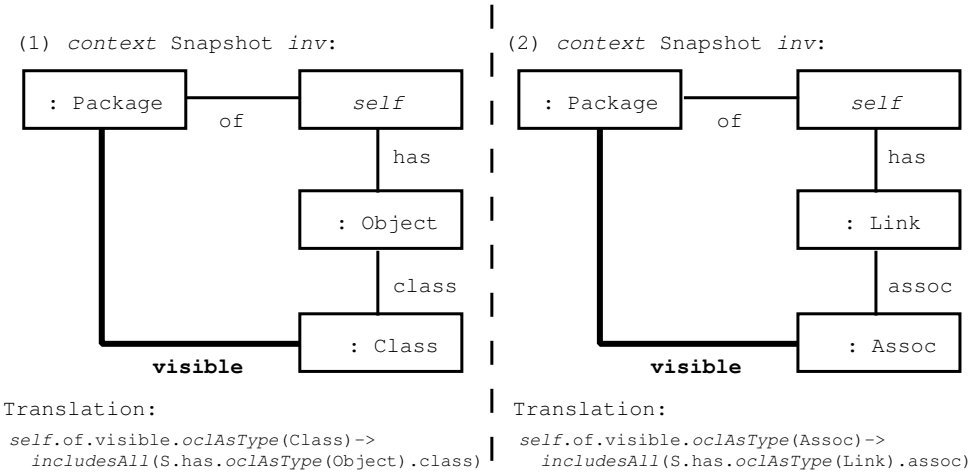

Fig. 7. Visual constraints defined as collaboration diagrams.

Figure 7 presents the definition of the first two OCL constraints of Figure 3, which are repeated at the bottom of the figure. The drawings have to be

[4] The dot in a spider diagram expresses existential quantification.
[5] The star in a spider diagram expresses universal quantification.

read as follows. For any `Snapshot` object `self` inspect all combinations of related `Package` and `Object` (`Link`) instances with their related `Class` (`Assoc`) instances. These matches of the nonbold parts of the depicted graph patterns must be extendible to matches including the bold parts of the given graph patterns. In this case the bold parts require the existence of a derived `visible` relationship between the matched `Class` (`Assoc`) instance and the matched `Package` instance.

Figure 8 shows a more elaborate example of a *collaboration constraint diagram*. It is equivalent to constraint (3) of Figure 3 repeated at the bottom of Figure 8. It requires for a `Link` between a `source` and a `target` `Object`, which is an instance of an `Assoc(iation)` between a certain `source` and a certain `target` `Class`, that the `Class` of the `source` (`target`) `Object` conforms to the `source` (`target`) `Class` of the `Assoc(iation)`.

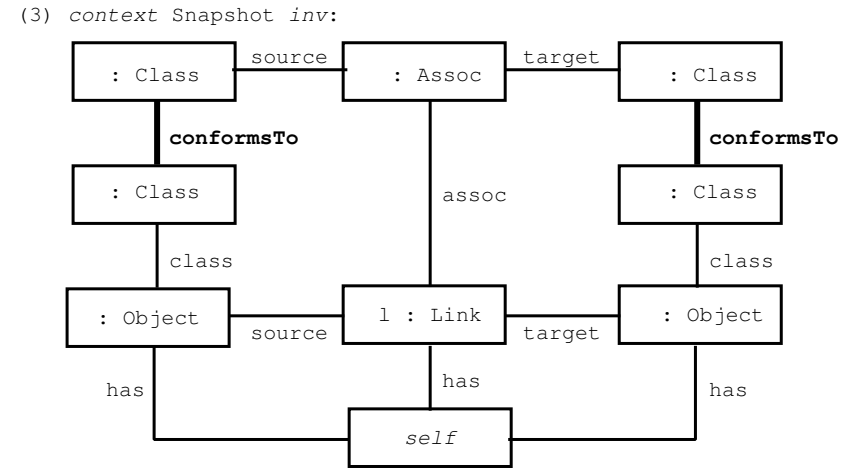

Fig. 8. A (more) complex visual constraint.

The proposal presented in [1] to use a variant of collaboration diagrams for the definition of constraints also supports some kind of visual negation. The "regular" part of a given diagram must be extendible to a match of its regular part plus its bold parts and must not be extendible to a match of its regular part (plus its bold parts) plus its dashed parts. This is the semantics for negative subgraph patterns used by the graph transformation system AGG. Relying on our experiences with the usage of negation in the graph transformation language PROGRES this interpretation of negative subpatterns seems to be nonintuitive under certain circumstances.

Regard for instance the visually defined new version of the derived relationship `visible` in Figure 9.[6] It determines all matches for the given two

[6] The idea of the visual specification of derived relationships based on graph patterns with

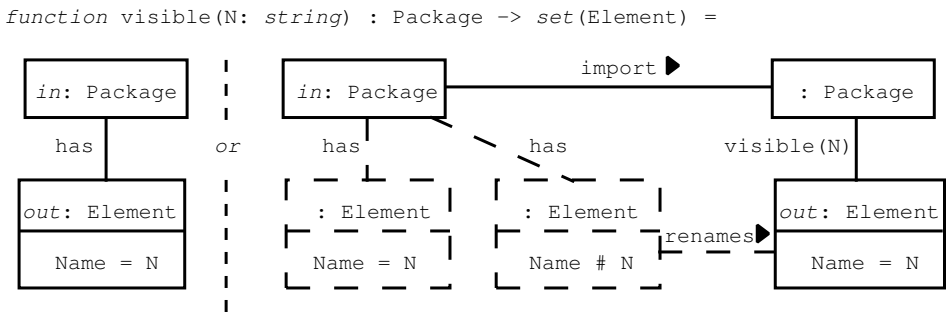

Fig. 9. Functional abstraction and negation for visual constraints.

subdiagrams, where the distinguished `in Package` matches the primary input argument of `visible`. It returns the set of all `Element` instances of all computed matches of nodes (objects) labeled with `out`. Applied to a given package it returns therefore its own components with the required name or some visible component of other packages with the proper name if certain conditions are fulfilled.

The AGG semantics for the right-hand side subdiagram of function `visible` excludes the case that the regarded package contains an element with the required name *and* that it renames an imported element with the proper name *at the same time*. This is not the condition we had in mind. What we want to express is the fact that an element with `Name = N` from a package `Q` is visible in a package `P` if `P` imports `Q` and *neither* `P` contains already an element with the same name *nor* renames the imported element to a different name.

To summarize, PROGRES requires that a match of all nodes and edges of a diagram with regular borders may not be extended to a match for all nodes and edges with regular borders plus a *nonempty subset* of its nodes with dashed borders plus their adjacent edges.[7] AGG, on the other hand, requires that a match of all nodes and edges of a diagram with regular borders may not be extended to a match for all nodes and edges with regular borders plus *all* its parts with dashed borders. Please note that the intended definition of function `visible` maybe expressed in the algebraic graph transformation approach by comstructing one (negative) extension for each dashed element node and its adjacent edges of Figure 9. In this case, a single diagram with one kind of dashed graph elements is no longer sufficient to write down the needed requirement.

Further discussions and experiments are needed to determine whether in most cases the AGG variant or the PROGRES variant behaves as expected by a software engineer, who is not a graph transformation system expert.

---

a fixed start node `in` and a fixed result node (set) `out` is adopted from PROGRES.

[7] The PROGRES syntax uses crossed-out nodes and edges instead of dashed nodes and edges for indicating negation.

5 Conclusions and Future Work

In this paper we first compared UML's object constraint language OCL with a similar sublanguage of the graph transformation language PROGRES and identified quite a number of deficiencies of the current version of the OCL standard. Based on our experiences with the development of PROGRES and the experiences of other people in the graph transformation community with incorporating graph transformation concepts into UML [12] or OCL [1] we presented a number of proposals for extending OCL.

A first group of proposed extensions concerns the introduction of some additional operators (for building transitive closure etc.), a second group concerns the addition of a visual constraint sublanguage based on collaboration diagrams. As far as we know based on the attendance of the OCL workshop associated with the UML'2000 conference no other proposals for adding similar operators have been made until now. The proposed operator `unique` is the only exception we are aware of. Adding such an operator to OCL version 2.0 was one of the points of discussion at the mentioned OCL workshop.

Concerning the proposed addition of a visual constraint language we have to emphasize that our proposal adopts the syntax proposed in [1], but suggests a different interpretation of negation and adds the concept of functional abstraction for visually defined constraints, too. Nevertheless, we have to admit that it is a matter of debate whether the textual versions of the OCL constraints or their visual counterparts presented here are easier to produce and to understand. But a more substantial discussion of this subject is useless as long as there are no tools for editing, analyzing, and executing visually defined constraints. To a certain extent all collaboration diagram editors of UML CASE tools can be used for editing the new visual constraints, but the appropriate support for analyzing and executing the constraint defining collaboration diagrams is still missing. Even worse, until now no commercial UML CASE tool, we are aware of, is able to analyze and execute standard OCL expressions.

It is the subject of ongoing activities of other research groups [10,15] to design and implement OCL processing tools. It is the subject for future research activities at our site to redesign the syntax of PROGRES path expressions such that it becomes compatible with a future OCL standard definition, while preserving its precise static and dynamic semantics definition and its integrated programming environment.

References

[1] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In *[7]*, pages 294–308, 2000.

[2] T. Clark. Typechecking UML static model. In *[9]*, pages 503–517, 1999.

[3] St. Cook, A. Kleppe, and R. Mitchell et al. The amsterdam manifesto on OCL. Technical report, 2000. http://www.trireme.com/amsterdam/manifesto-1-5.pdf (visited: 11/07/2000).

[4] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.

[5] C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment. In *[4]*, pages 551–605. 1999.

[6] L. Euler. Lettre a une princesse d'allemagne. 2(102-108), 1761.

[7] A. Evans and St. Kent, editors. *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, volume 1939 of *Lecture Notes in Computer Science*, Berlin, 2000. Springer Verlag.

[8] UML Revision Task Force. OMG unified modeling language specification v. 1.3, document ad/99-06-08. Technical report, Object Management Group, 2000. http://www.omg.org/uml/ (visited: 07/11/2000).

[9] R. France and B. Rumpe, editors. *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, Berlin, 1999. Springer Verlag.

[10] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *[7]*, pages 278–293, 2000.

[11] St. Kent. Mixing visual and textual constraint languages. In *[9]*, pages 384–398, 1999.

[12] U. Nickel, J. Niere, and A. Zndorf. The Fujaba environment. In *Proc. ICSE 2000 - The 22nd Int. Conf. on Software Engineering*, pages 742–745. ACM Press, 2000.

[13] The precise UML group. PUML home page. Technical report, 2000. http://www.cs.york.ac.uk/puml/ (visited: 11/07/2000).

[14] M. Richters and M. Gogolla. A metamodel for ocl. In *[9]*, pages 156–171, 1999.

[15] M. Richters and M. Gogolla. Validating UML models and OCL constraints. In *9*, pages 265–277, 2000.

[16] Andy Schürr. New type checking rules for OCL (collection) expressions. 2001. submitted to GI-Workshop Modellierung'2001.

[17] Andy Schürr, Andreas J. Winter, and Albert Zündorf. PROGRES: Language and environment. In *[4]*, pages 487–550. 1999.

[18] J. Venn. On the diagrammatic and mechanical representation of propositions and reasoning. *Phil. Mag.*, 123, 1880.

[19] J. Warmer and A. Kleppe. *OCL: The Object Constraint Language - Precise Modeling with UML*. Addison Wesley, New York, 1999.