A Visualization of OCL using Collaborations *

Paolo Bottoni[1], Manuel Koch[2],
Francesco Parisi-Presicce[1], Gabriele Taentzer[3]
[1] Università di Roma “La Sapienza Italy - [2] PSI Berlin -
[3] Technical University of Berlin

Abstract. We propose a visualization of OCL within the context of the UML meta model, so that OCL expressions are represented by extending collaboration diagrams. We exploit the OCL meta model introduced in [9] and further elaborated on in [1] and base the description of properties of objects on collaborations, while classifier and association roles are used to describe navigation paths. Operations computing properties are described by interactions consisting of messages between classifier roles. The introduction of new graphical core elements is kept to a minimum. New notation mainly concerns the predefined operations in OCL and provides more convenient visual forms for the notation by interactions here. The proposed visualization is described in detail and is illustrated with examples taken from an industrial project under development.

1 Introduction

Although UML is gaining acceptance as a standard for designing and documenting software and systems, there are still wide differences as to the use of its components. In particular, OCL is still of limited use even in organizations which extensively employ some form of UML diagrams. The reasons for this are various, but they can often be traced back to the difficulty of integrating a purely textual language like OCL into diagrams. Recommendations have been issued towards the formulation of a diagrammatic version of OCL [3] and some proposals have been advanced in order to provide a visual counterpart of OCL by exploiting visualizations with a set-theoretic semantics [5]. The disadvantage of such proposals is that the diagrams look quite different from others in UML and thus, force the user to learn yet another visual language. Moreover, such new visualizations are often not well integrated in the meta model of UML.

The work presented in this paper aims at achieving a visualization of OCL within the context of the UML meta model while departing as little as possible from the use of graphical elements already belonging to the UML notation. Where a new notation is adopted, it is easily recognizable as derived from the UML core notation, or it is simply employed as a shortcut to an equivalent more complex visualization. Additional meaning is attached to some specific notation and to a selected set of spatial relations and aspectual features of the notation.

This visualization adapts the notation for UML core elements. Furthermore, we follow the recommendations of the standards, by avoiding the use of colors

* Partially supported by the EC under Esprit Working Group APPLIGRAPH.

or special typography to express meaning. Throughout the paper, we illustrate the principles behind our choices for the proposed visualization and expand on how these principles led to specific choices. A prominent principle is to limit the introduction of new graphical elements. This responds to two main motivations. On the one hand, it limits the cognitive effort of the user, thus facilitating diagram understanding. On the other hand, the use of elements already in the UML syntax facilitates the checking of model diagrams against the constraints expressed by our OCL diagrams. Another principle aimed at better understanding is to limit the size of the diagram. In particular, no two occurrences of a sub-diagram need be present in the same diagram. Moreover, visual shortcuts for predefined operations support a compact notation of OCL constraints.

In [2] we proposed to use graph rewriting with rule expressions to provide a semantics for OCL, exploiting graph rewriting mechanisms for checking the consistency of model diagrams with respect to constraints. The use of a graph-based formalism opened the way to a visualization of OCL constraints and an extension of the UML meta model was defined to this effect by introducing two new classes. Now, we proceed towards a full description of a visualized OCL, by defining a meta model for OCL which combines the proposal in [9] with the UML meta model for collaborations. Hence, OCL constraints can be expressed in a more visual way, reducing the need for additional text.

The rest of the paper is structured as follows: the following subsection introduces the running example taken from an industrial project and used to illustrate the visualization of OCL constraints throughout the paper. Sections 2, 3 and 4 present the visualization proposed for OCL, showing sample OCL constraints in textual and visual form. In Section 5, we adapt the OCL meta model proposed in [9] and further elaborated in [1] to the visualization proposed. Finally, Sections 6 and 7 discuss related work and conclusions.

1.1 The running example

Our example is taken from an 'eGovernment' project in Berlin on which one of the authors has worked. The project objective is to replace the existing software system used in the residents' registration office in Berlin by a new software system that supports and facilitates both the business processes within one or among several authorities and the business processes between the authority and the citizen by exploiting Internet technologies. The main responsibilities of the residents' registration office are the registration of inhabitants, the maintenance and preparation of the inhabitants data for other authorities like police or fire department, and the certification of passports and ID cards.

The client requires explicitly the use of object-oriented technologies. UML is used, as it is the standard object-oriented notation also accepted by the client. However, whereas class diagrams, collaboration and sequence diagrams are techniques widely used in discussions about specifications between the client and the designer, OCL does not attract interest at all despite its appropriateness. Constraints, if any, are expressed in natural language.

The experience acquired in this project and in other big OO-projects in the author's company point to a simple conclusion: OCL is too difficult not only for the client but also for the designer and does not facilitate the transition from the intended requirements to a precise and legible presentation. Natural language formulations are preferred, though their ambiguity might often cause misunderstanding later in the project.

In the sequel, we present some of the constraints occurring in the project. Fig. 1 shows a (very small) part of the business object model of the project for which constraints have been defined. We give a brief explanation of the parts that are not self-explanatory.

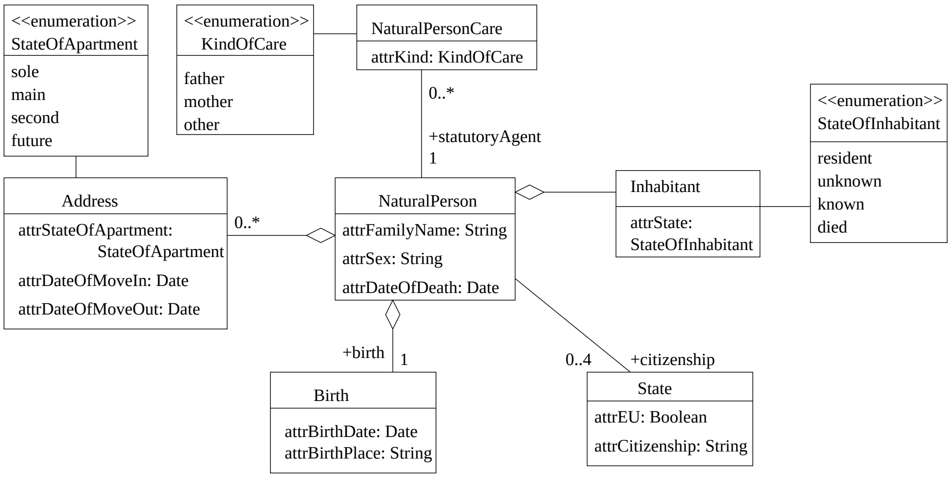

Fig. 1. *Business Object Model.*

The Class `NaturalPerson` describes natural persons (as opposed to legal persons). These are persons currently registered in Berlin or previously registered (e.g. moved outside Berlin or dead) or non resident members of the family. The class `Address` contains addresses of natural persons. The possible attribute values of the attribute `attrStateOfAppartment` are: <u>sole</u> apartment, <u>main</u> apartment, <u>second</u> apartment, <u>future</u> apartment (to which the inhabitant will move), and the <u>apartment</u> (to which the inhabitant has moved). The class `NaturalPersonCare` contains the custody, care or custodianship, as a statutory agent, of a natural person. By default, the mother becomes the statutory agent of a child. Additionally, the father becomes the statutory agent of a child if the father's family name is the child's family name. The values for the attribute `attrKind` are: child's <u>father</u>, child's <u>mother</u>, and <u>other</u>. The class `Inhabitant` contains data of natural persons who are inhabitants. For instance, non resident or dead natural persons are not inhabitants. The values of the attribute `attrState` are: <u>resident</u> natural person, non resident natural person with unknown address, non resident natural person with known address, and natural person who died in Berlin.

2 Constraints on Properties of Objects based on Collaborations

One of the main ingredients of OCL are navigation expressions to object properties. The visualization of navigation paths helps the developer to maintain an overview without forcing a reformulation of an object structure, given in some static structure diagram, in a completely different syntax. Thus, the idea is to base object navigation on collaboration diagrams.

Conforming to the meta model presented in [9], we consider that all OCL expressions produce a value. The type of this value can either be a data type or a classifier type of some instance. Expressions producing values are presented within an enclosing frame where the value is in a precise position with respect to the frame. If the value is of type Boolean, the explicit mention of the value can be omitted, provided that no other reference to it is needed. Hence a framed diagram without an external return value maps to a Boolean value. If the OCL expression produces an instance, or a collection of instances, the corresponding object or multi-object is double-bordered. Such an element must be reachable by navigating the diagram from the instance defining the context of the expression (usually named by 'self'). The kind of constraint is indicated by shortcuts 'inv', 'pre' or 'post' (for invariants, pre- or postconditions, respectively) in the upper left corner of the visualized constraint.

2.1 Object Attributes

The following is a simple OCL constraint stating that the birth date of a natural person comes before the date of moving into an apartment; the constraint is stated textually in the usual OCL syntax.

```
context NaturalPerson inv:
self.birth.attrBirthDate < self.address.attrDateOfMoveIn
```

The visualized form of this constraint, in Figure 2, contains three classifier roles, two with attributes. The attribute values, `x` and `y`, are compared in a simple Boolean constraint in the bottom compartment.

2.2 Operations and Methods

In OCL, operations and methods may be used if they are queries, i.e. they are free from side-effects.

Although in the project constraints for operations and methods have not been explicitly formulated using OCL syntax, such constraints are known to the designers. For instance, we have constraints for the range of the possible values of

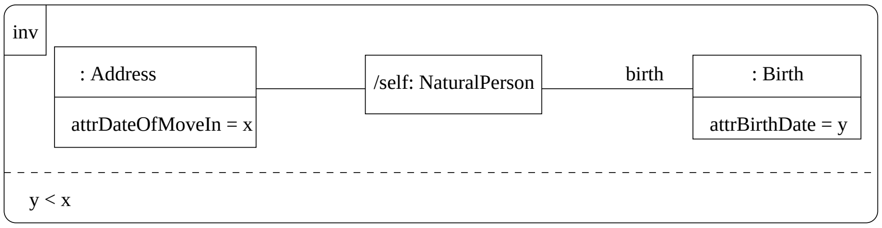

Fig. 2. *The birth date comes before the date of moving into an apartment.*

the attributes. Consider Figure 3 for the visualization of a constraint saying that the date of birth has to be earlier than today. Similar constraints are necessary also for the other dates occurring in the business model.

```
context NaturalPersonCare inv:
self.birth.attrDateOfBirth < currentDate()
```

This constraint is visualized by an instance of the self classifier role. A method call may be depicted as an interaction with this instance as the target. Its return value can be used to formulate a constraint. As a shortcut, the method compartment can contain the constraint, as shown in Figure 3.

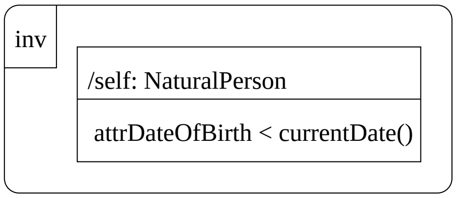

Fig. 3. *The date of birth has to be earlier than today.*

2.3 Navigation through associations

This part of an OCL constraint is the most interesting one to be visualized. Navigation through an object structure is much easier if the path can be visualized directly. Consider again the example in Figure 2. The constraint visualization shows three classifier roles, with two association roles in between, where a role name is attached to one. A more complex navigation is shown in Figure 8.

If the multiplicity of the association end has a maximum of 1, the target is an object, otherwise a multi-object. Examples for navigation to multi-objects will be shown in Section 3, where the target is a collection. When describing the navigation in this visualized form, role names are mandatory only if there is more than one association between two classes.

Navigation from association classes is similar to usual navigation. Since the navigation from an association class to one of the objects in the association always results in one object, we never have multi-objects as source or target objects of the association. If the associations are qualified, also these qualifiers can be used for navigation. The qualifier is denoted via the usual UML notation.

3 Collections and Operations on Collections

OCL supports three different types of collections: sets, bags and sequences. They are given in the UML_OCL package in an abstract way, i.e. element types are left open. For their concrete usage, the abstract element type has to be instantiated. The visualization may stick to the usual representation of classifier roles but may also use special notions to support the three predefined collection types.

These special notions of sets corresponds directly to the layout of multi objects in UML, and the correspondence extends to the semantical level as well. We call this visual form *set box*. The other types of collections are represented by adorning the set box with details recalling the collection type. Hence, a *sequence* is represented by connecting the corners of the shifted rectangles with dots recalling a series, and a *bag* is represented by placing a semi circle, recalling a handle, over the upper rectangle of the collection element (see Figure 4). The concrete element type of a collection occurs inside the rectangle in front.

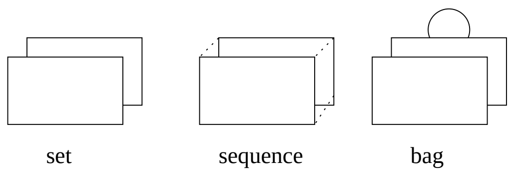

Fig. 4. Visualization of collection types

3.1 Query Operations

Collections offer a variety of query operations which support the computation of projections based on boolean expressions. Applying such a projective query operation to a collection results in a new collection (or in a scalar).

So far, we described the application of an operation by an interaction. This is possible for query operations, too, but these predefined operations can be visualized more adequately based on set boxes.

In the following, we take the `select` operation to describe how query operations and their application can be visualized. If the selecting expression is simple, i.e. a constraint on attributes, this constraint may be included into the constrained classifier role as a shortcut. Usually, the selecting expression is framed by a set box. Such a representation is employed on the left of Figure 5 to select all the addresses of a natural person who is a resident or a non-resident with a known address. The number of addresses has to be greater than 0. The expression `#n` inside the dotted box produces a name for the cardinality of the selected collection. Moreover, the iterating variable may be depicted on the right side of the set box (here 'a'). This visualization of a query is very flexible, since there might be a complex expression visualized within the set box. The visualization becomes difficult if the set of addresses might be used in other queries as well. In this case, there is a preference for the visualization on the right of Figure 5. Here, a special *isIn* edge is introduced which expresses the fact that the addresses in the selecting expression belong to the set of addresses of a natural person. Moreover, the natural person is repeated in the selecting expression, as depicted by an id-connection. The corresponding textual OCL constraint looks like:

context `NaturalPerson` inv: self.address→select(naturalPerson.inhabitant.attrState = #resident or naturalPerson.inhabitant.attrState = #known)→ size > 0

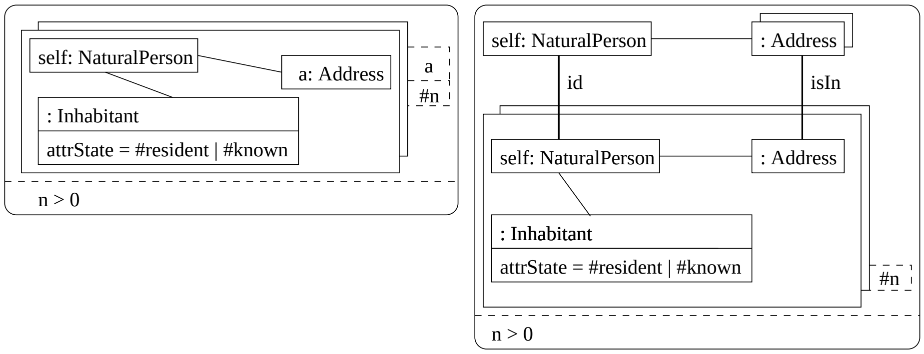

Fig. 5. A select statement with two variants

The use of the alternative operator (|) in the figures is a shortcut for a disjunction involving duplication of the classifier role to present the possible alternative values.

An overview of all visualized query operations for collections is given in Figure 6. For a better understanding, we show the visualization of the *collect* operation together with the conversion to a certain collection type. Actually, the conversion to a bag is not necessary, since the resulting collection is always a bag. The *iterate* operation is slightly more complicated, since it contains the specification of an iterator *elem* and an accumulator *acc*. These are variable declarations depicted on the right side. The accumulator gets an initial value which is depicted by *init* in the left upper corner. Please remember these visual notations are meant as shortcuts for pure collaboration diagrams with queries being interactions.

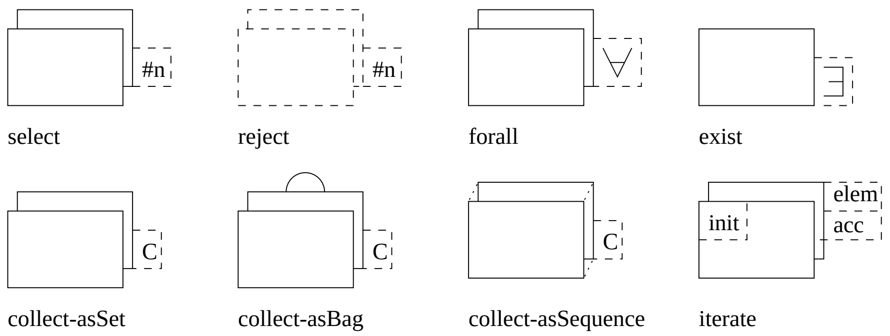

Fig. 6. Visualization of query operations

3.2 Further Operations on Collections

OCL predefines many operations on collections. In the following, we briefly discuss the shortcuts proposed for the main ones. The operation *size* has already been used, visualized by a hash mark and a variable name at the right border of a collection frame. (Consider Figure 5.)

The operation *includes* can be visualized by an arrow from the element to the set which has to include this element. The arrow is labeled by *includes*.

The visualization of the predicate *isEmpty* is done by presenting the tested collection in a dashed style. This visualization should stress the non-existence of any instance. For the same reason, all adjacent links should be drawn dashed as well. In general, negated parts of a diagram are drawn with dashed lines.

Due to space limitations, we do not discuss shortcuts for all predefined operations of collections. They follow the same design principles as the ones we discussed.

4 Visualized Logical Expressions

Logical expressions can occur at two different levels: 1) Expressions on simple data types, denoted textually in the usual OCL syntax; 2) Logical expressions on object navigation, represented by object interaction or more adequately by several sorts of frames and dashed diagram parts for negation.

As an example of the first type, the textual requirement "There are no instances of class `Address` for natural persons without a state of inhabitation", when formulated in natural language, is ambiguous. The intended meaning is that the attribute `attrState` has the value "unknown" or "died", but it could also be interpreted to mean that the attribute has no value at all. The OCL syntax, in this case, helps disambiguate the request:

```
context NaturalPerson inv:
    self.inhabitant.attrState = #unknown or self.inhabitant.attrState = #died
implies
        self.address→isEmpty
```

The conclusion contains some redundancy, since it repeats the classifier role "self: NaturalPerson" which is needed to formulate the conclusion. The visual constraint is depicted in Figure 7.

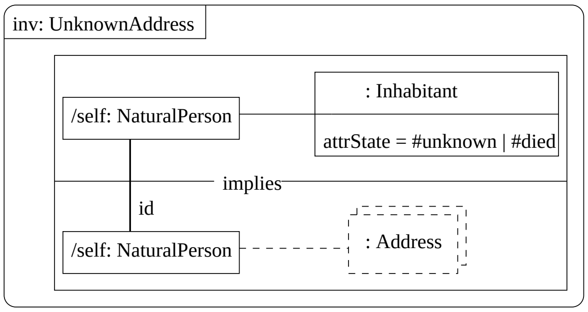

Fig. 7. *There are no instances of class Address for natural persons without a state of inhabitant.*

As an example of the second type, the visualization for "If the statutory agent is *mother*, the birth date of the statutory agent must be earlier than the birth date of the child" is shown in Figure 8.

```
context NaturalPerson inv:
     (self.naturalPersonCare.attrKind = #mother)
implies self.birth.attrBirthDate <
      self.naturalPersonCare.statutoryAgent.birth.attrBirthDate
```

For the formulation of the constraint "The citizenships of a person are different", "we need a "for all" query: Its visualization is depicted in Figure 9.

```
context NaturalPerson inv:
self.citizenship− >forall(x, y : state| x.attrCitizenship = y.attrCitizenship
implies x = y)
```

The use of identical classifier roles allows a form of factorization, or of abstraction, used in several places in our visualization. In fact, in a UML expression, elements can be referenced several times. This can occur explicitly as in *let* expressions, or implicitly if a navigation expression is textually replicated

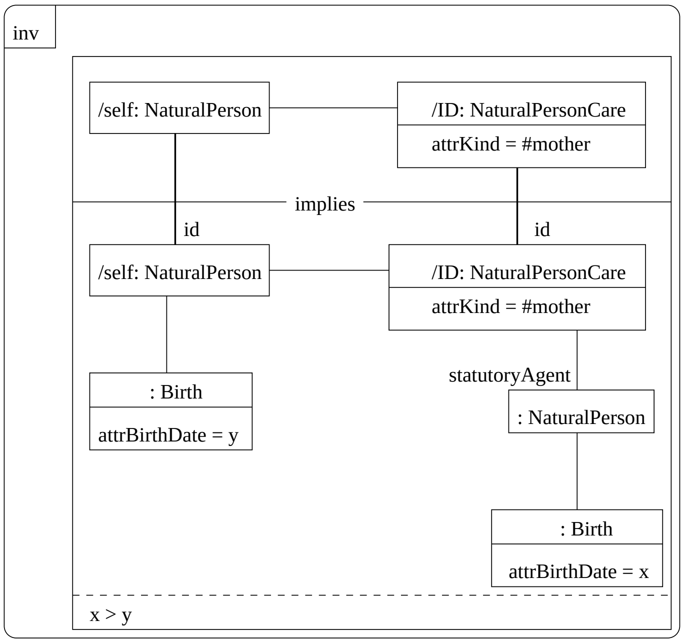

Fig. 8. *If the statutory agent is* mother*, the birth date of the statutory agent must precede the birth date of the child.*

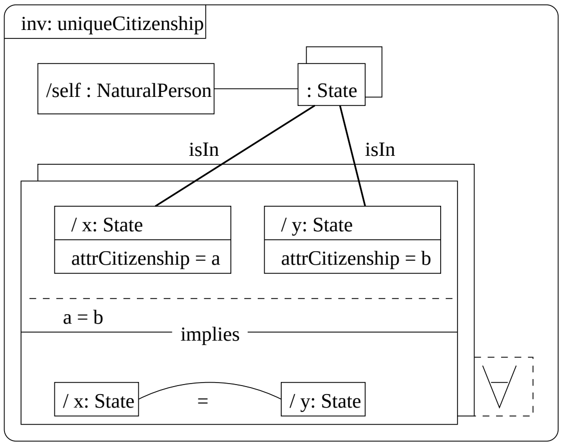

Fig. 9. *The citizenships of a person are different.*

in different parts of a textual constraint. In the latter case, an element corresponding to the longest common prefix of a set of navigation expressions can be identified. In any case, we present only one model element and produce another box connected to the original element for each reference to it in the diagram. If a named element, e.g. an instance of a classifier, is replicated, then identity may alternatively be expressed by the use of the same name.

A similar mechanism can be used to decompose a diagram into subdiagrams, via a hierarchical view of the expression. Boxes in a high level diagram can represent whole subdiagrams. In this case, the box can be named and expanded in different subdiagrams. This can be used whenever a complex expression is used several times. An example is shown in Fig. 10 which contains the conjunction of the constraints shown in Fig. 7 and 9.

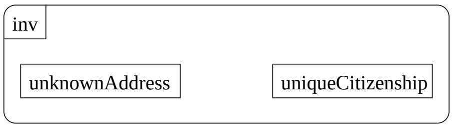

Fig. 10. A composed constraint.

Boolean predefined operations are represented by associations labeled by the name of the operation. Boolean connectives are represented by combining appearance of graphical elements with their enclosing and nesting.

In particular, negation, disjunction and conjunction are treated. We elaborate on the convention of Peircean graphs [4] which represent conjunction by copresence in the diagram and negation by enclosing the negated elements in a contour. On the contrary, we represent negation by modifying the appearance of the elements (using dashed lines) and use confinement within contours to express both conjunction and disjunction. In particular, an odd value of the nesting level indicates that the expressions within a same frame are taken in conjunction and an even value indicates that the expressions are taken in disjunction.

More precisely, we assume that the logical expressions are organized as *AND* / *OR* trees, with an *AND* connective at the root. Each subtree is enclosed within a frame and its depth determines whether it has to be interpreted as a conjunction or as a disjunction. Expressions within a frame at an *AND* level of nesting are taken as conjuncts, while expressions within a frame at an *OR* level of nesting are taken as disjuncts. In case the original OCL formula had disjunctions at the top level, a new fictitious top node is first inserted to constitute the *AND*-labeled root, and then the translation process is started. In this case, the first UML element can be found at the second layer of nesting at least. A dashed frame indicates that the whole expression within it must be negated, while dashed individual elements indicate that they enter the expression in a negated form.

If-then-else expressions can be depicted by frames with different compartments. The *if* compartment is above the *then* compartment on the left, and the *else* compartment on the right. A sample constraint like “there is a date of death if the person has died” looks as shown in Figure 11.

```
context Inhabitant inv:
if self.attrState = #died then

        not(isUndefined(self.naturalPerson.attrDateOfDeath))
else
        isUndefined(self.naturalPerson.attrDateOfDeath)
```

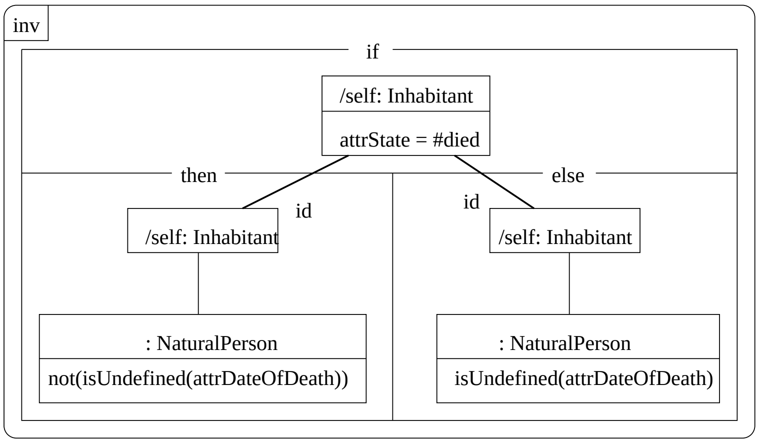

Fig. 11. *There is a date of death if the person has died.*

5 Adaptation of the OCL Meta model

Basing the visualization of OCL on collaborations, the OCL meta model introduced in [9] and further elaborated in [1] has to be adapted to the meta model for collaborations. The idea is to use collaborations to describe properties of objects. In this sense, roughly speaking, the sub-meta model for a property operation is replaced by a section of the meta model for collaborations.

The description of object properties is based on classifier and association roles which are used to describe navigation paths. Here, the existence of navigation paths is expressed by the usual collaborations whereas non-existence can be expressed by negated classifier and association roles. A special classifier role is the self classifier role. The dynamic aspects of collaboration diagrams are exploited to represent the calling of methods to determine properties. Method calls are described by interactions consisting of messages between two classifier roles. These messages are actions, possibly with parameters.

Attribute expressions can be described by collaborations too, since classifier roles might contain attributes with instantiation. The proper constraint is expressed by a Boolean expression attached to the property operation.

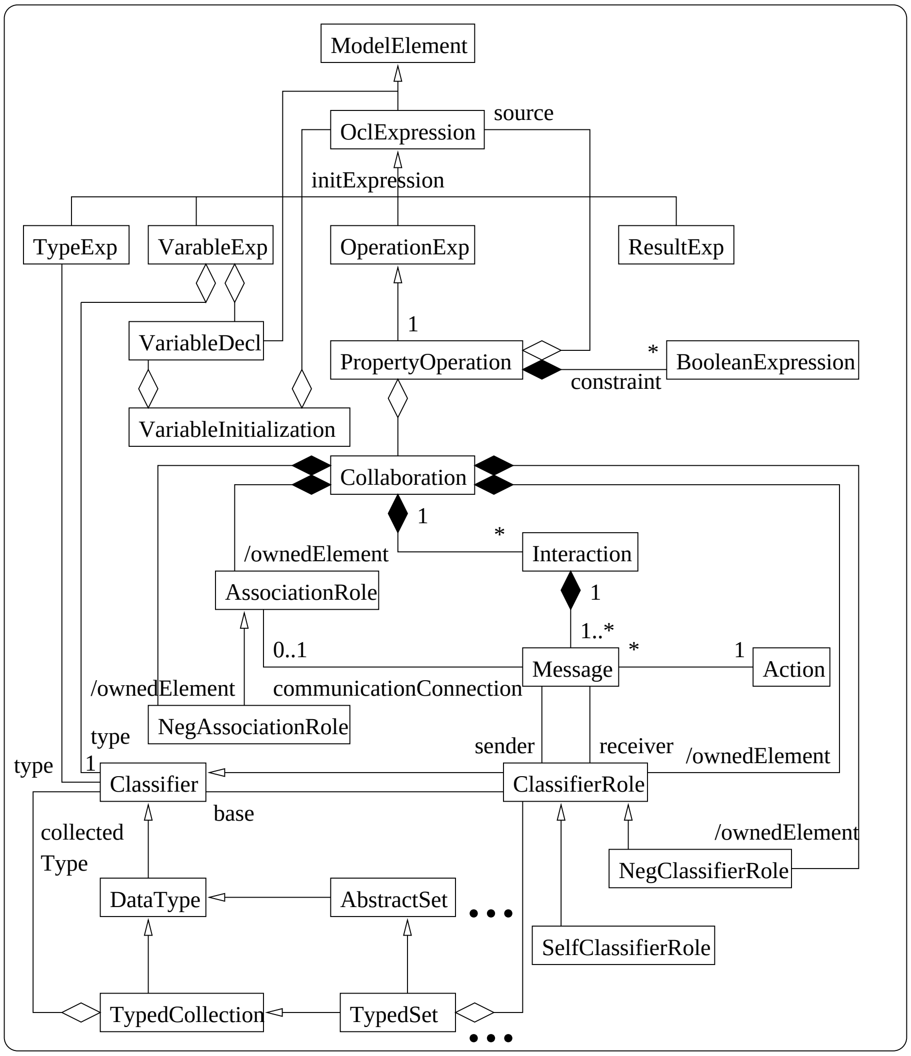

Fig. 12. Meta model for Visual OCL

As in [1], a special package UML_OCL contains basic data types and collections. Abstract collections are thought to be incorporated in this package as data types. They have to be instantiated by concrete element types, which is done in a special profile which introduces typed collections with a link back to classifiers to capture the collected type.

Operation expressions are property operations in any case. If they consist of constraints only, simple Boolean expressions are formed as constraints. Special OCL operations are integrated by offering special data types in the UML_OCL package as described above. As already presented in [9], OCL expressions may be variables, operations or result expressions. Self expressions no longer need to be distinguished, since self classifier roles are now introduced. Moreover, we do not need query expressions anymore, since their operations are now handled by the collection types in the UML_OCL package, as stated in [1]. Finally, type expressions, as introduced in [1], specify types which are needed in some operations based on the OCLAny type.

We do not have space to discuss additional constraints on the meta model.

6 Related Work

Constraint diagrams [7] and their evolution, spider diagrams [5], have been proposed as a way to express a set-theoretic semantics of OCL constraints. However, they do not represent a visualization of OCL expressions by supporting a structural correspondence between visual and textual constructs. Moreover, the approach is not based on the graphical elements already present in the UML core. The difficulty of immediately relating constraint diagrams to UML model diagrams suggested the combination of annotating such diagrams with fragments of textual OCL constraints defining the context of the evaluation, the environment defined by a let expression, or specific operations [8]. Such an approach suffers from the typical difficulties of parsing together diagrams and text, that we overcome by providing suitable visualizations for all the structural elements where [8] needs textual constraints, and by constraining the use of text in well defined positions in the diagram.

Following the OCL meta model proposed in [9], in [10] an approach is proposed for the validation of UML models and OCL constraints based on the animation of model diagrams. Static checks can be performed automatically, while the designer can interactively generate prototypical instances of a model in the form of diagram snapshots, to be compared against the specification. Although providing an interactive form of integration between diagrams and OCL constraints, the proposal is based on a purely textual syntax, obtained through a translation of UML to a textual representation and adding "wrapping" constructs on top of OCL syntax.

Discussing the general requirements for the construction of tools supporting OCL, Hussmann *et al.*, advocate the use of normal forms, such as the reduction of `forall` and `exists` constructs to `iterate` [6]. The existence of normal forms would facilitate the choice of alternative representations for constraints with the same models, e.g. `size = 0` and `isEmpty`. In this paper we have maintained adherence to the original syntax rather than to the semantics, as in Figure 6.

7 Conclusion

The paper has presented an approach to the visualization of OCL constraints based on collaborations, aiming at the formulation of complete constraining expressions on UML models in the same visual language as the model. The proposed visualization is based on a number of principles which favor a greater readability and amenability to OCL constraints. The proposal conforms to the UML meta model and adapts recent proposals for a meta model for OCL. The proposed visualization introduces a limited amount of new core notation, but offers a variety of visual shortcuts for convenient visual notation. They exploit spatial relations between model elements and their appearance properties in a consistent way. In particular, spatial relations are used in the form of frames with nesting to represent the tree structure of a Boolean expression.

As to appearance properties, we use different kinds of line thickness, replication for representing collections, adornments to qualify collections, and connecting lines or names, to express references to elements.

As future work, two mapping algorithms, commuting between textual and visual OCL notations, are desirable once the discussions on the OCL meta model have converged. In this case, the textual and the visual notations could be considered as two concrete syntaxes of one and the same abstract syntax.

References

1. M. Bodenmüller. The OCL Metamodel and the UML-OCL package. Proc. of OCL Workshop, Satellite Event of UML 2000, York, October 2000, 2000.
2. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Automatic Consistency Checking and Visualization of OCL Constraints. In *UML 2000*, pages 294–308. Springer LNCS 1939, 2000.
3. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and Wills A. The Amsterdam Manifesto on OCL. Technical Report tum-19925. Technical report, Tecnische Universität München, 1999.
4. E. Hammer. Peircean graphs for propositional logic. In G. Allwein and J. Barwise, editors, *Logical Reasoning with Diagrams*, pages 129–147. Oxford University Press, 1996.
5. J. Howse, F. Molina, J. Taylor, S. Kent, and J. Gil. Spider diagrams: A diagrammatic reasoning system. *Journal of Visual Languages and Computing*, pages 299–324, 2001.
6. M. Hussmannn, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and Selic B., editors, *UML 2000*, pages 278 – 293. Springer LNCS 1939, 2000.
7. S. Kent. Constraint diagrams: Visualising invariants in object oriented models. In *Proceedings of OOPSLA'97*. ACM Press, 1997.
8. S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *UML'99*, pages 384 – 398. Springer LNCS 1723, 1999.
9. M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99*, pages 156 – 171. Springer LNCS 1723, 1999.
10. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000*, pages 265 – 277. Springer LNCS 1939, 2000.