Working on OCL with graph transformations *

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

Abstract. The Object Constraint Language (OCL) provides an important complement to visual formalisms used in the definition of UML languages. Yet, its usage is limited by two major drawbacks. The first is the limited availability of tools for the automatic verification of constraints against model diagrams. The second is the difficulty of amalgamating a textual formalism such as OCL with the visual languages used in the rest of UML. We attack these problems with methods deriving from a graph-transformation based approach. We propose a visualisation of OCL, based on a recently proposed metamodel for it, which provides a declarative way to represent OCL constraints, and then we discuss how an operational semantics for OCL can be based on transformation units which guide the application of graph-transformation rules.

1 Introduction

The Object Constraint Language (OCL) provides an important complement to visual formalisms used in the definition of UML languages. It is used both as a textual counterpart to UML models, and to constrain extensions of UML using metamodels. Yet, its usage is limited by two major drawbacks. The first is the limited availability of tools for the automatic verification of constraints against model diagrams. The second is the difficulty of amalgamating a textual formalism such as OCL with the visual languages used in the rest of UML.

Both problems can be attacked based on the existence of some formal semantics for OCL. In particular, the existence of a metamodel, as introduced in [RG99], allows us to devise some form of visualisation, conforming to the metamodel, but which is better suited to integration with the UML diagrammatic languages. A visual formalism such as that of graph rewriting rules can also be used to provide an operational semantics for OCL, which can be applied both to its visual and to its textual representation. Such an operational semantics can be realised on top of existing tools for graph rewriting.

The paper first presents the main concepts behind the proposed visualisation and then discusses how an operational semantics can be achieved in terms of transformation units defining a strategy in the application of graph transformation rules. The conclusions illustrate some benefit of the approach

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

2 Visualisation of OCL Constraints

In the following, we consider two main issues when visualizing OCL. One is the visualization of navigation expressions, ubiquitous in UML. Another important issue is the visualization of collections and their operations. The visualization concepts for both issues are illustrated by examples taken from an industrial project on 'E-Government'. The project objective is to replace the existing software system used in the residents' offices in Berlin by a new software system that supports and facilitates both the business process 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 underlying business object model contains classes like `NaturalPerson` which describes natural persons as opposed to legal persons, and `Inhabitant` containing data of natural persons being inhabitants. Additional constraints must be checked to insure the consistency of the data base with the intended application. A more detailed presentation of the business model can be found in [BKPPT01]. In the same paper, the interested reader can also find a discussion on the principles behind the proposed visualization.

2.1 Constraints on Navigation Expressions

OCL constraints involve complex navigation expressions to reach object properties. A user trying to follow these expressions incurs in the cognitive cost of having to reformulate an object structure, given in some static structure diagram, in a different, textual, syntax. Hence, visualizing navigation paths would help the developer to maintain an overview of the structure while reasoning about the constraints. We propose to express object navigation in a visual, declarative way through collaboration diagrams.

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 1, contains three classifier roles, two of which present attributes. The attribute values, `x` and `y`, are compared in the bottom compartment. The kind of constraint is indicated by shortcuts 'inv', 'pre' or 'post' (for invariants, pre- or postconditions, respectively) in the upper left corner. Four alternative versions for visualizing this constraint could be employed, as navigation to `Birth` and `Address` objects can be performed on named as well as on anonymous associations.

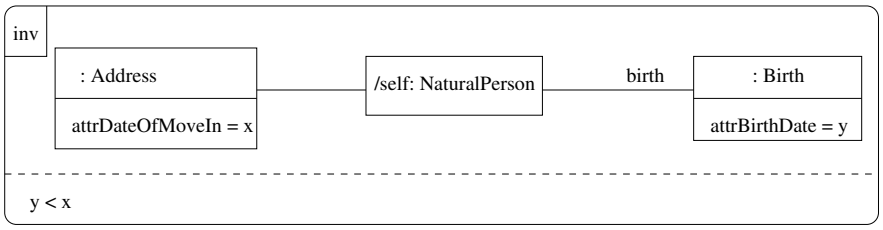

Fig. 1. *The birth date comes before the date of moving into an apartment.*

2.2 Collections and Their Operations

OCL supports three different types of collections: sets, bags and sequences. Sets are represented using the convention for multi-objects in UML. Under this convention, a multi-object represents all the objects, possibly an empty collection, reached by a navigation expression. 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, i.e. connecting the corners of the shifted rectangles with dots (to remind of a series) for a *sequence*, and placing a semi circle, reminding a handle, over the upper rectangle of the collection element for a *bag*. The concrete element type of a collection is in the front rectangle. Basing the visualization on collaborations, the application of an operation is described by an interaction.

We use the `select` operation to describe how a collection operation can be visualized. Usually, the selecting expression is framed by a set box. Such a representation is employed in Figure 2 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 must be greater than 0. The corresponding textual OCL constraint is:

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

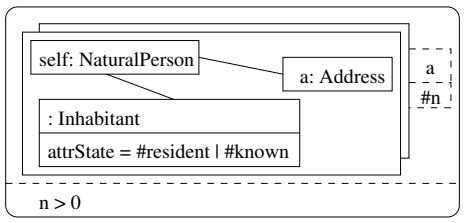

Fig. 2. A select statement

2.3 Logical expressions

The use of the alternative operator (|) in Figure 2 is a shortcut for a disjunction involving duplication of the classifier role to present the possible alternative values. A more general visualization has been devised for logical expressions. In particular, logical expressions on object navigation are represented by framing expressions in order to reproduce the nesting of logical AND and OR operators, where each level is alternately read as an AND or an OR, starting from an outermost AND frame. 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. Dashed diagram parts are used for negation.

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.

2.4 OCL Metamodel

As we base the visualization of OCL on collaborations, we perform some adaptation of the OCL meta model introduced in [RG99] and further elaborated in [Bod00] to make it consistent with the meta model for collaborations. The idea is to use collaborations to describe properties of objects. This is natural, since the description of object properties is based on classifier and association roles which are used to describe navigation paths. The dynamic aspects of collaborations are exploited to represent the calling of methods to determine object properties.

As in [Bod00], 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. Special OCL operations are integrated by offering special data types in the UML_OCL package as described above.

3 Graph Transformations for Checking Constraints

The main motivation to develop OCL has been the definition of well-formedness rules in the context of the UML semantics, but it may also be used for precise modeling of user applications. To express OCL semantics by graph transformation, a function $tr : Set(OclExpression) \rightarrow Rules \cup Set(TransformationUnit)$ is defined. An OCL constraint is an expression, with a boolean return value, satisfied by an instance model, if the corresponding rule or transformation unit can be applied to the instance graph. The evaluation of the rule or unit does not modify the graph on which the constraint is checked.

3.1 The Graph Transformation Approach

We work on directed, typed, and attributed graphs. Rule application follows the single-pushout approach to graph transformation [Löw93,EHK+96]. A rule

may also contain a set of *negative application conditions (NAC)* to express that something *must not* exist for a rule to be applicable [HHT96]. The negative condition can refer to values of attributes [TFKV99]. Rules can also employ *set nodes*, which can be mapped to any number of nodes in the host graph, including zero. The matching of a set node is in any case exhaustive of all the nodes in the host graph satisfying the condition indicated by the rule. Set nodes have to be preserved between the left and right context, but new set nodes can be created. This implies that it is not possible to use a rule to destroy a set of nodes, but the nodes in the set have to be removed individually from it. Finally, set nodes must not occur in NAC's.

Transformation units are used to further control rule application. In the following example, the denotation of transformation units is mainly reduced to the containing rules and control conditions. Initial and terminal configurations are all instances of the given UML model. The import relation of units remains implicit. The control condition is specified by expressions over rules.

Given a set $Names$ of rule names from which rule expressions are constructed, a rule expression E as we will use it, is a term generated by the following syntax:

- basic operators:
  $E ::= Names \mid E_1$ **and** $E_2 \mid E_1$ **or** $E_2 \mid E_1; E_2 \mid$ **a**$(E) \mid$ **na**$(E) \mid$
  null $\mid$ **if** E_1 **then** E_2 **else** E_3 **end** $\mid$ **while** E_1 **do** E_2 **end**
- derived operators:
  $E ::= E_1$ **implies** $E_2 \mid E_1 \ = \ E_2 \mid E_1$ **xor** $E_2 \mid$ **asLongAsPossible** E **end**

Most of the operators presented above have the obvious meaning: operators **a** and **na** test the applicability and non-applicability, resp. Each rule expression is either applicable or non-applicable, i.e. it has a boolean return value. In case, it is applicable, the unit it controls can also produce a value, a node, or a set node, according to the declaration of the arguments for the unit. In the operators **if-then-else** and **while-do-end**, the rule expression E_1 is tested for applicability (without being applied). The result of this test determines how to proceed with application in the usual way. Operator **asLongasPossible** applies a rule expression to a graph as long as it is applicable. A detailed formal definition of rule expressions as presented above can be found in [BKPPT00].

3.2 Checking simple OCL constraints

An OCL constraint over navigation expressions can be translated into a graph rule, easily derivable from the visualization of such OCL constraints by static collaboration diagrams. Such diagrams can be interpreted as identical graph rules, i.e. both sides are equal and the rule morphism is the identity (e.g. the OCL constraint in Figure 1 which can be interpreted as an identical graph rule).

If a rule which is the translation of a constraint can be applied to some instance graph, the constraint is satisfied for this instance. The non-applicability of a rule can have two causes: either there is no total mapping of the left-hand side to the instance graph or no mapping satisfies all the additional application

conditions. In both cases, this can be reported to the user helping him/her to find the inconsistency. After the user has performed further editing steps on the UML model, the checking can be started again. Thus, the rule-based character of inconsistency checking could be advantageously used to perform constraint checking any time the user so wishes.

3.3 Checking advanced OCL constraints

The proposed visualisation does not always have a direct procedural counterpart and direct matching is not sufficient when some strategy has to be followed in constraint checking. For example, checking a constraint expressed as a nesting of logical connectives, or navigation expressions involving a universal quantifier requires following a precise sequence of actions.

The translation from OCL constraints to rule expressions can be performed systematically, as the translation to the declarative visualisation. We consider here again the example in Figure 2. We assume that the predefined operations, such as `select`, have already been translated into suitable transformation units. Then, the whole constraint is translated again into a transformation unit which uses the select unit.

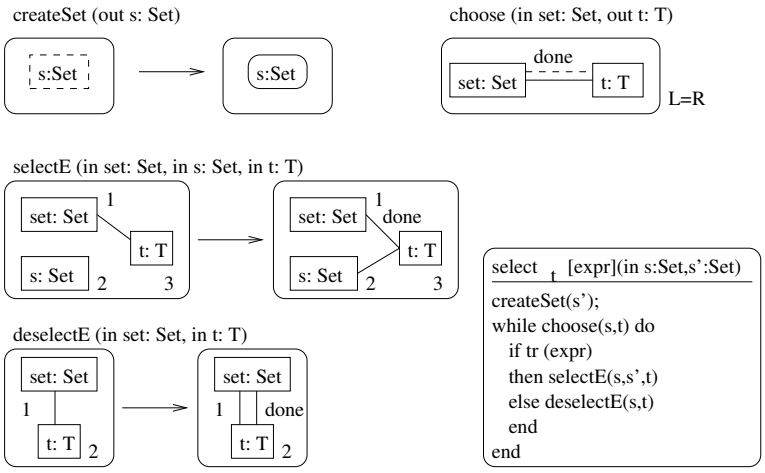

Fig. 3. Translation of pre-defined operation `select`

The `select` operation is directly dependent on the evaluation of the selecting expression. The selecting expression is part of the resulting transformation unit, thus the unit name is dependent of the expression. (Consider the translation of `select` in Figure 3.) In the body of this transformation unit, the operation is first initialized by creating an additional structure (here a `Set` node) and then the select action is performed. Note that the dotted box in the left-hand side

of rule `createSet` indicates that the set s must not be already present in the diagram for the rule to be applicable. For each element chosen from the given set (and not already visited, i.e. without an adjacent `done` edge), we have to select or deselect it, depending on the selecting expression. The chosen element is used as input parameter of the selecting expression.

Having the translation of `select` available, the translation of the OCL constraint in Figure 2 is depicted in Figure 4. The first rule navigates to all the addresses. Then two transformation units are called, performing the selection of addresses of residents or known persons and counting them. The last rule checks if there is at least one of those addresses.

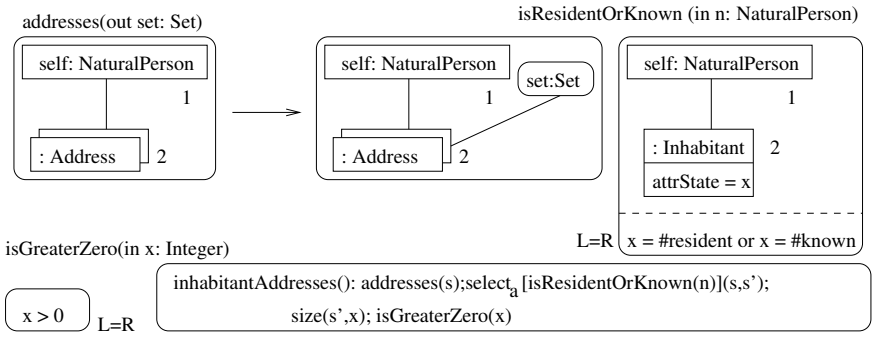

Fig. 4. Translation of OCL constraint in Figure 2

During a constraint check, the instance graph may be augmented by additional objects and links. These objects, which may be collection nodes and additional `done` edges, have to be deleted by additional rules collected in a special transformation unit which is invoked after each check.

3.4 An example on an instance diagram

As an example of consistency check of an instance diagram, consider the object diagram in Figure 5, showing a portion of the data base in the E-Government project. We want to check the OCL constraint in Figure 2 on the two instances of `NaturalPerson` there.

To test the well-formedness of a diagram or model, we have to look for the applicability of a set of rules and/or transformation units. Looking at our sample model in Fig. 5, the transformation unit in Fig. 4 can be applied fully to the portion of the diagram relative to Manuel, but not to the portion relative to Gabi. Indeed, the transformation unit in Figure 4 will start by constructing the set of addresses, which results empty for Gabi. Now the `select` transformation unit will start by constructing the new set to accommodate the selected address, but it will not enter the `while` loop, as the rule defining the looping condition is not applicable. Hence, the `select` transformation unit will end by returning

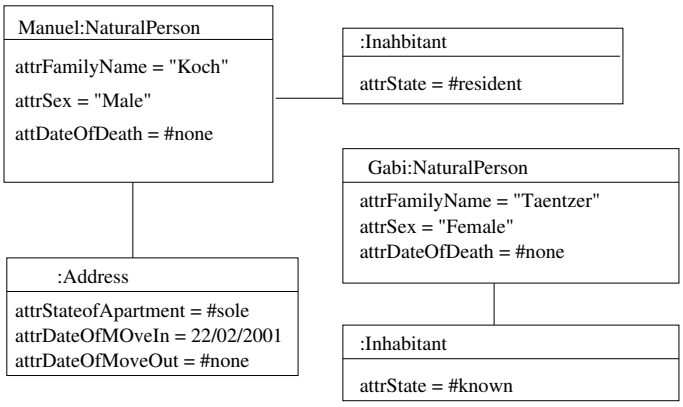

Fig. 5. An object diagram not satisfying a constraint

an empty set. The `inhabitantAddresses` transformation unit will then proceed to compute the size of this set and finally the rule for `isGreaterZero` is not applicable, making the whole unit to fail. Conversely, when applying the unit to Manuel's portion, the selected set contains the only address present for Manuel, so that it is not empty, and its size is greater than zero, testifying to that the constraint is satisfied.

3.5 Discussion

The identification of the transformation units translating a textual OCL constraint can proceed in parallel with the translation to a visual counterpart, providing an operational reading of the declarative visual representation. The two visual forms of management of OCL constraints, i.e. visual OCL constraints and graph transformations, share some commonalities at the base level, presenting elements from the UML syntax, namely, classifiers, associations, multi-objects and attributes. However, the composition of complex constraints is expressed through ad hoc visualisations in one case and through rule sequences in the other. In general, rule expressions can be employed to govern the application of rules for complex constraints, in a way which can be inferred by the structure of the visualisation. Conversely, the visualisation provides suitable graphical constructs to express the iteration and alternative operators of rule sequences.

Using graph transformation to give OCL an operational semantics is rather closely related to the dynamic metamodeling approach by Engels *et al.* who provide a graphical approach to the operational semantics of behavioural UML diagrams [EHHS00]. To this end, they use collaboration diagrams, which can be interpreted as graph rules. The two approaches mainly differ in the way rule application is controlled. While in the metamodeling approach rule applications can be controlled by sequential and parallel composition as well as the usage of other transformation units, we allow a larger variety of control constructs.

4 Conclusions

We have proposed to exploit forms of visualisation to achieve a smoother integration and use of OCL in the context of the UML diagrammatic languages, both from a declarative and from a procedural point of view. To this end the paper has illustrated how visual representation of constraints exploiting the UML visual syntax can be achieved, and how graph rules can be used to support consistency checking of OCL constraints on target diagrams. Two systematic translations from the textual OCL syntax to the two forms of visual syntax (static visualization and executable graph transformation units) preserving the semantics of the constraints can be realized. These visual syntaxes admit some amount of hybridization with textual syntaxes, as conditions on properties or primitive OCL operators (such as `size` or `isOCLKindOf`) are more simply left in the textual form (see for instance the realization of isGreaterZero in Figure 4). The presented visualisation is based on collaborations, and is consistent with the metamodel for OCL proposed in[RG99]. It introduces a limited amount of new core notation, but offers a variety of visual shortcuts for convenient visual notation, favouring a greater readability and amenability to reasoning of OCL constraints. The combination of the visualisation of OCL and the application of rule expressions has the advantage of allowing an intuitive representation to the user, who can perform direct checking on the model, and of getting a formal semantics, in terms of transformation units.

Based on the proposed translation, an OCL evaluator can be implemented on top of a graph transformation machine like AGG or PROGRES ([EEKR99]) and later integrated into a UML CASE tool. These tools support a step by step evolution of the underlying host graphs. An OCL evaluator based on such a graph transformation machine can help to understand the implemented OCL semantics by following the stepwise evaluation on instance diagrams visually. An editor for visual OCL constraints is currently being implemented for the open source CASE tool ArgoUML. The graph transformation-based approach to checking inconsistencies can easily support automatic repair actions, by defining suitable graph rules to solve them, if possible. This approach relies on the idea of living with inconsistencies during software development presented in [GMT99], also on the basis of graph transformation.

References

[BKPPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans and S. Kent, editors, *UML 2000*, number 1939 in LNCS, pages 294–308. Springer, 2000.

[BKPPT01] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In M. Gogolla and C. Kobrin, editors, *UML 2001*, number 1939 in LNCS, pages 257–271. Springer, 2001.

[Bod00] M. Bodenmüller. The OCL Metamodel and the UML-OCL package. Proc. of OCL Workshop, Satellite Event of UML 2000, York, October 2000, 2000.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[EHHS00] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In A. Evans and S. Kent, editors, *UML 2000*, number 1939 in LNCS, pages 323–337. Springer, 2000.

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 247–312. World Scientific, 1996.

[GMT99] M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland*. IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[HHT96] A. Habel, R. Heckel, and G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae*, 26(3,4), 1996.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *TCS*, 109:181–224, 1993.

[RG99] 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.

[TFKV99] G. Taentzer, I. Fischer, M. Koch, and V. Volle. Visual Design of Distributed Systems by Graph Transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*, pages 269–340. World Scientific, 1999.