Consistency Checking and Visualization of OCL Constraints*

Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, Gabriele Taentzer**

Università di Roma, Italy, {bottoni,carr,parisi,gabi}@dsi.uniroma1.it

Abstract. Part of the success of the Unified Modeling Language (UML) as a specification language is due to its diagrammatic nature. Its meaning is expressed by its meta model, a combination of class diagrams and constraints written in the Object Constraint Language (OCL), a textual language of expressions. Recent efforts have tried to give a formal semantics to OCL in a classical way. In this paper, we propose a graph-based semantics for OCL and a systematic translation of OCL constraints into expressions over graph rules. Besides providing a semantical formalization of OCL, this translation can be employed to check the consistency of UML model instances wrt. the constraints , using a general purpose graph transformation machine like AGG or PROGRES. The translation of OCL constraints into graph rules suggests a way to express the constraints in a more intuitive visual form.

1 Introduction

The meta model for UML is not only defined by class diagrams, but also by well-formedness rules, used to give a more precise abstract syntax for UML diagrams. Well-formedness rules are stated by OCL constraints.

Up to now, the concrete syntax of OCL and an informal semantics are given in [WK98]. Recently, also a meta model has been proposed for OCL [RG99]. Previous efforts by the same authors [RG98] have produced a semantics for OCL expressions in a classical way. First a function from sorts and sort expressions to standard semantical domains is defined inductively; then, an expression is interpreted as a function from variable assignments and states to the semantic domain of the appropriate sort. What is not clear in their approach is how this formal semantics can be used to establish the consistency of a model diagram with respect to a constraint. Since the interpretation of OCL is in set theoretic terms, the validation of a constraint still needs the integration of text and diagrams. Such an integration can be more easily achieved in a graph transformation setting, with its ability to manage any type of relation among multi-dimensional structures. To achieve this, we first show how OCL constraints can be translated into graph rule expressions. Evaluating these expressions for certain UML instances means checking their consistency wrt. corresponding OCL constraints,

* Research partly supported by the TMR network GETGRATS, and the ESPRIT Basic Research Working Group APPLIGRAPH

** On the leave of TU Berlin, Germany, gabi@cs.tu-berlin.de

being these either well-formedness rules or user-defined model constraints. Consistency checking can thus be automated by exploiting existing tool support for graph transformations. The approach is used to check well-formedness rules on UML models or diagrams created and edited during the software development process.

The translation of OCL constraints into expressions over graph rules indicates a way to visualize some of the OCL constraints. Collaboration diagrams can be used to visualize navigation expressions within OCL. Embedding them into logical expressions written textually leads to a hybrid notation of OCL constraints, which is more suitable for UML models than the pure textual one, since objects and navigation expressions are depicted in the same way as in UML.

In this paper, Section 2 presents the main concepts of graph transformation informally (see the formalization in [Roz97]). Graph transformations are applied in Section 3 to define a formal semantics for OCL. Section 4 discusses a new hybrid notation for OCL and compares it to related work.

2 Graph Transformation

In this section, the main notions of graph, graph rule and its application, and transformation unit are briefly reviewed, describing only informally the notation used and referring to the literature for the formal definitions and results.

The nodes and arcs of a graph may be typed and attributed, where attributes are specified by a type, a name and a value. Each graph node and arc may have several attributes.

Graph rules are used to describe graph transformations and contain a left-hand side and a right-hand side. The graphs occuring in a rule are typed and attributed: left-hand sides are allowed to have variables used to abstract the operation from concrete attribute values. The left and the right-hand side of a rule are related by a partial graph mapping $L \to R$ which specifies the correspondence of graph objects by numerical tags.

Furthermore, rules may have parameters which are useful to determine e.g. matches and attributes of new graph objects by the user. A rule may contain a set of *negative application conditions (NAC)* which are able to express that some graph part *must not* exist for a rule to be applicable. Basically, this is realized by adding to the rule another dashed graph which represents the part that must not occur.

Finally, *set nodes* may appear in the rule, meaning that they can be mapped to any number of nodes in the host graph, including zero. Set nodes have to be preserved and must not occur in NAC's. They are represented as double rectangles.

The way graph rules are applied realizes directly the algebraic approach to graph transformation as presented in [Roz97]. The combination of attributed graph transformation with negative application conditions has been worked out in [TFKV99].

Rule application is performed in two steps: First, the applicability of a rule is checked by looking for a match $m : L \to G$ of the left-hand side into a host graph G. A match is a total mapping, i.e. each graph object of L is embedded into the graph G. If a variable occurs several times in a rule's left-hand side, it must be matched with the same value. Note that in general, there may be multiple matches of the rule's left-hand side into the host graph, or there may be no matches at all. In the latter case, the rule is *not applicable* to the given host graph. A rule is applicable at a certain match, if all its NAC's and further attribute conditions are satisfied.

In the second step, the matching pattern found for the rule's left-hand side is taken out of the host graph and replaced by an appropriate matching pattern for the rule's right-hand side. Since a match is a total mapping, any object o of the rule's left-hand side L has a proper image object $m(o)$ in the host graph G. Now if o has an image $r(o)$ in the rule's right-hand side R, its corresponding object $m(o)$ in the host graph is *preserved* during the transformation, otherwise it is *removed*. Objects in R which are not the image of an object in L are *newly created* during the transformation. Finally, the objects of the host graph which are not covered by the match are not affected by the rule application at all.

Besides manipulating the nodes and arcs of a graph, a graph rule may also perform computations on the object's attributes. During rule application, expressions are evaluated with respect to the variable instantiation induced by the actual match.

Transformation units (in [EEKR99]) provide the means to structure a system of rules into graph procedures that can use each other. Within a unit, the set of rules of another unit can be 'called' by using the name of the corresponding unit, thus realizing functional abstraction. More formally, a transformation unit consists of

- an expression I, that defines a set of graphs $SEM(I)$ to be given as input,
- an expression T, that defines a set of graphs $SEM(T)$, produced as output,
- a set R of rules,
- a control condition C, describing a binary relation $SEM(C)$ on graphs, and
- a set U of imported transformation units represented by a set of identifiers.

Semantically, a transformation unit specifies a binary relation from initial configurations in $SEM(I)$ to terminal configurations in $SEM(T)$. The "interleaving" semantics contains a pair of graphs (G, G') if $G \in SEM(I)$, $G' \in SEM(T)$ and G can be transformed into G' by using rules in R and the calls to the imported units according to the control condition C possibly after the unit has been preprocessed to adapt it to the context. The control condition C is specified with expressions over rules as described next.

A rule expression E, built up over a certain set of the operators, specifies a set of sequences of rule names, denoted by $seq(E)$. If one can apply to a graph the rules in the order given by one such sequence, the sequence is *applicable* to the graph and the graph is transformed into the *expression graph*. A rule expression is applicable to a graph, if at least one sequence in $seq(E)$ is applicable to it.

Given a set $Names$ of rule names from which rule expressions are constructed, a rule expression E 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 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 [BKPT00].

3 Translation of OCL Constraints

The OCL has been designed as an extension of UML to model constraints in a more precise way [WK98]. The main motivation to develop OCL has been its usage for defining well-formedness rules in the context of the UML semantics, but it may also be used for precise modeling of user applications. Throughout this paper, we consider version 1.3 of OCL [UML99] and restrict to invariants, but the ideas can also be applied to pre and post conditions. All examples are taken from the UML semantics description [UML99].

To express OCL semantics, we define a function $tr : Set(OclExpression) \rightarrow Rules \cup Set(TransformationUnit)$. An OCL constraint is an expression with a boolean return value. It is satisfied by an instance model, if the rule or transformation unit to which it is translated can be applied to the instance graph. The evaluation of the resulting rule or unit does not modify the instance graph on which the OCL constraint is checked. In the following examples, 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.

3.1 Expressions on object properties and simple OCL operations

An expression on object properties is an attribute expression, a navigation expression or a classifier operation. Such an expression is translated into a rule containing extended static collaboration diagrams as left and right hand sides. The extensions deal with the specifics of rules containing a mapping between the rule sides, set nodes and application conditions. The name of the rule is chosen according to either the last role name in the navigation path, or the attribute or the constraint to be tested.

Attribute expressions: Given an OCL expression that produces a result of a non-boolean type, a variable of this type is created by the obtained rule. The rule is named by the attribute name. It has an input parameter being the name of the object node to which the attribute belongs. But if the expression is the first one in a constraint, the rule does not have an input parameter. The rule itself consists of an object node of input parameter type or context type together with the required attribute. The left constraint in Fig. 1 consists of an attribute expression only, thus there is no input parameter. Since the attribute is of type Boolean, the rule does not have an output parameter, too. Furthermore, in this case the two rule sides are identical, as expressed by $L = R$.

Navigation expressions: As for attribute expressions, there is a node of the result type of the source expression in the rule. At this node, an edge together with another node are added, representing a linked object, if the multiplicity is 0 or 1. The edge contains the role name and possibly qualifiers used for navigation. The second node is of the type the corresponding association points to. In case of multiplicity greater than 1, a set node of this target type is in the rule and a node of type `Set` and links to all the elements (depicted by the set node) are created. `Set` nodes are depicted by rounded rectangles meaning that they have a special attribute that indicates a restricted life time. They live only during the checking process and are deleted afterwards (Compare Subsection 3.6.) If the association is ordered, not a `Set` node but a `Sequence` node is created instead. Its outgoing edges carry the information about the ordering. The rule name is chosen according to the role name. The input parameters are handled as for attribute expressions, the output parameter points to the target object or collection.

Classifier operation: These operations are directly translated into attribute computations, if they are queries. The rule, named by the operation name has the source as well as the arguments as input, and the result value, computed as value of an attribute variable, as output parameter. If the result is an object, this object has to occur in the rule graphically. If it is newly created, it occurs only on the right-hand side, otherwise on both sides. If an operation has a boolean result, this is expressed by the applicability of the corresponding rule.

Simple OCL operations: OCL operations on simple types, e.g. $+$, $-$, $=$, and $\geq$, are translated into attribute computation, exactly as classifier operations.

The right constraint in Fig. 1 contains an OCL operation with an attribute expression as argument. This would normally lead to two rules applied one after the other. In the figure, the already sequentially composed rule is depicted. In general, a translation of complex expressions over object properties which results in a sequence of rules can be condensed into one rule by building the composed rule. (Compare also rule `isGreaterAggregation` in Fig. 5.)

If a non-boolean operation or method has to be performed (which is still a query), it is called on the right-hand side and is assigned to an attribute or output parameter to be usable in following rule applications.

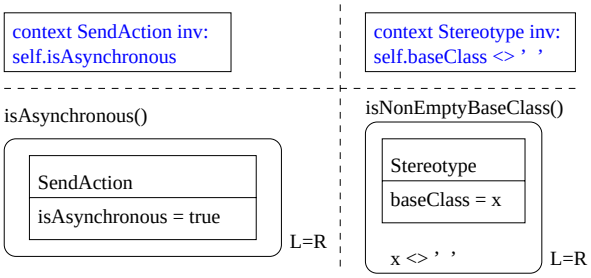

Fig. 1. Simple OCL constraints and their translations

3.2 Types

OCL has a type hierarchy which is translated into a special node attribute, called `type`, of type `OclType`. For all those types there is a partial order allowing a matching of nodes not only if their types are equal, but also if the matching is compatible with this order. In Figure 2, two interesting properties of `OclType` and `OclAny`, i.e. `allInstances` and `oclIsKindOf`, are translated into two rules. Note that in the left rule, variable `type` has to be instantiated by a type instead of an instance as is usual. Further on, a `Set` node is created. The usage of `allInstances` is not recommended [CKMR99], since the context of application is not clear. This translation clarifies the situation in the sense that the set of instance graphs to which the rule is applied is the given context.

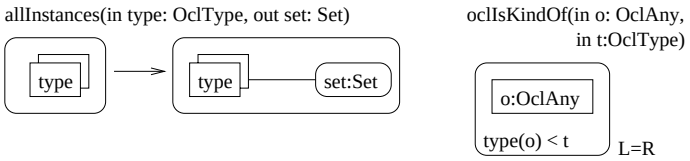

Fig. 2. Translation of two interesting properties of `OclType` and `OclAny`

Primitive values and types are translated into suitable attributes. These are types such as `Real, Integer and String`. Moreover, type `Boolean` belongs to this groups as long as boolean expressions are considered on primitive values only. Instances of non-basic predefined types such as `Collections` and their subtypes `Set, Bag` and `Sequence` as well as the types considered in the given UML model, are translated into corresponding nodes together with a set of links to associated objects. Moreover, we have to consider the type `Boolean` once again, if the original OCL expression contains boolean navigation expressions. This is done in Subsection 3.4.

Predefined types have predefined operations. Predefined operations on collections (sets) such as *sum* or *size* are translated into rules and transformation units controlling the rule application (compare Figures 3 and 4). Several rules are needed to collect information on a variable environment of the context class to compute the operation's result. Additional nodes and edges used by called rules store intermediate information and are deleted after checking the whole constraint. In the following paragraphs, we consider some of the pre-defined types and operations more closely.

3.3 Collections

Navigation taking predefined collection operations into account is performed by rules or transformation units. The result type of a navigation expression depends on the operation: if it is not boolean, there is an output parameter pointing to the result. A boolean result is treated differently, i.e. implicitly. It is expressed by the applicability of the refining rule expression.

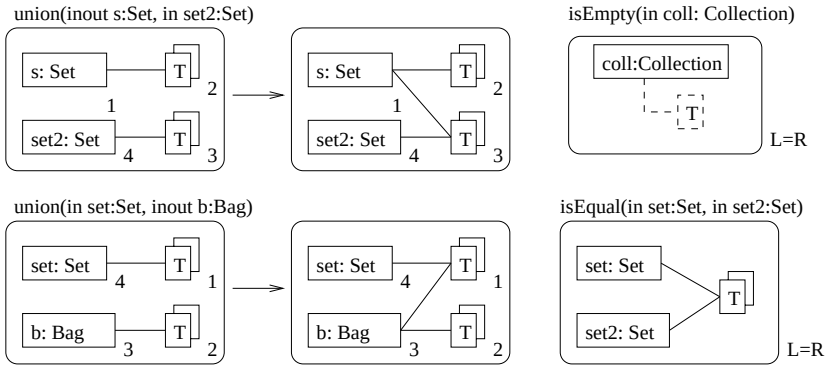

Fig. 3. Translations of sample pre-defined operations on collections

Figure 3 shows some sample operations on collections translated into graph rules. Whereas the operations `union, isEqual` and `isEmpty` can be described directly by one rule, operations like `select` and `size` have to be described by transformation units. The translation of `select` is presented in Figure 4. Due to space limitations, we do not show the translation of all the pre-defined operations, but a number of examples that the reader may complete.

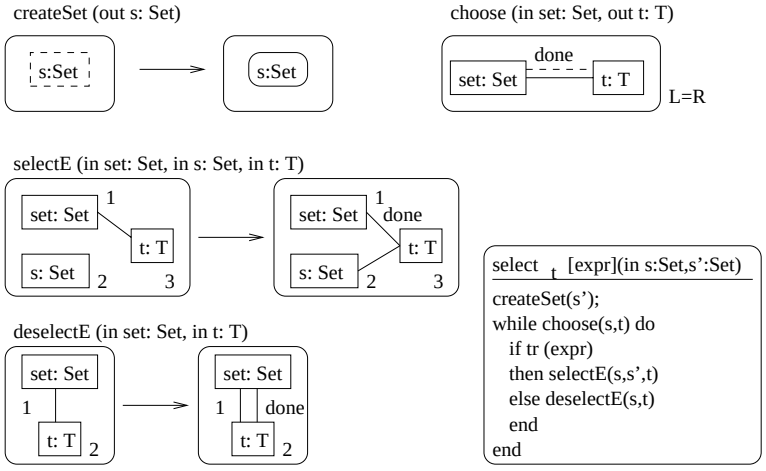

Fig. 4. Translation of pre-defined operation `select`

Select operation: Another interesting pre-defined operation for sets is `select`, since it 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 4.) In the body of this transformation unit, first the operation is initialized by creating additional structure (here a `Set` node) and then the select action is performed. 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.

If the selecting expression is an attribute or a subtype expression, it is not necessary to first choose an element, but the evaluation of the selecting expression can be incorporated directly into the select and deselect rules.

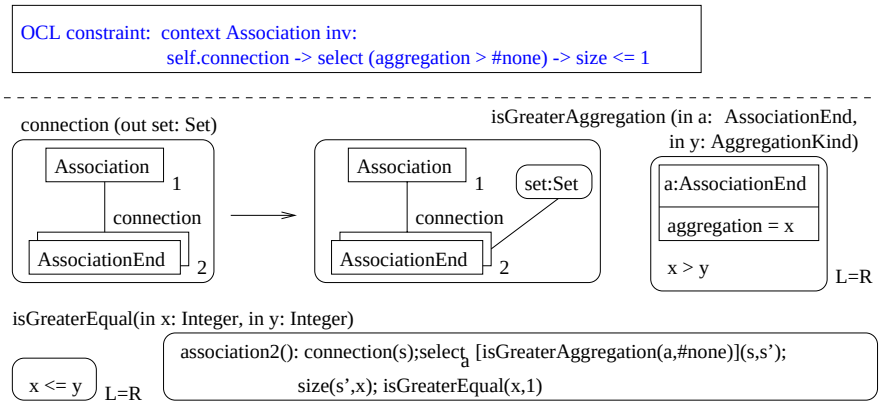

Fig. 5. Translation of a well-formedness rule for associations

Having the translation of `select` available, we consider another well-formedness rule for UML, this time one for associations given at the top of Figure 5. It states that at most one association end may be an aggregation or composition. A direct translation of this OCL constraint leads to a transformation unit performing four rule applications and unit calls sequentially. The first rule navigates to all the association ends. Then two transformation units are called, performing the selection of association ends which are aggregations or compositions and counting those. The last rule just checks if there is at most one of those association ends.

3.4 Boolean expressions

Boolean expressions on OCL expressions are translated directly into rule expressions. For each boolean OCL operator, there is a corresponding operator for rule expressions.

The well-formedness rule for name spaces depicted in Figure 6 together with its translation, uses boolean operators frequently. This constraint states for each

element which is contained in the name space and is not an association or generalization that it has a unique name. The translation uses a transformation unit `forall`$_{m1,m2}$`[expr](in s:Set)` checking an expression for all elements of a set `s`. The body contains a while-loop checking the expression for each element similarly to the body of `select`. Rule `isEqual` enforces a mapping of both model elements on exactly one.

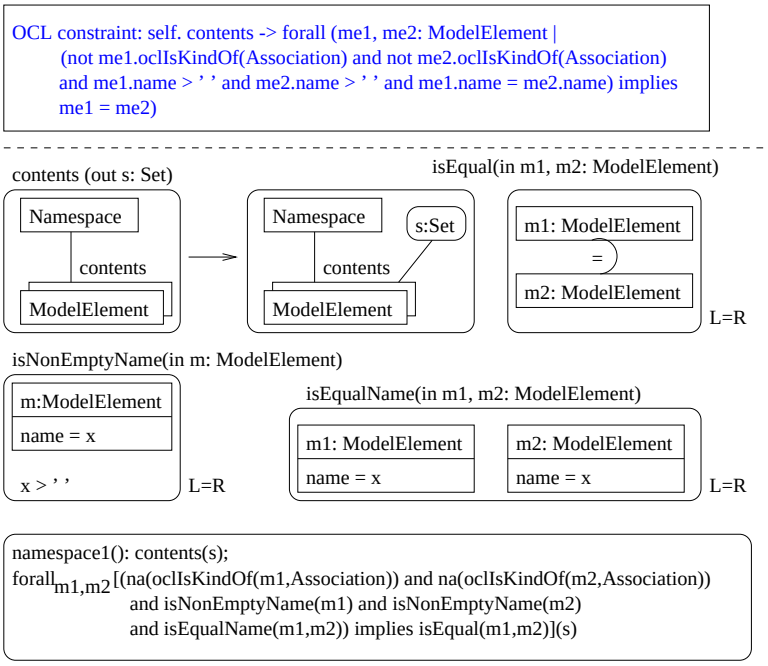

Fig. 6. Translation of a well-formedness rule for name spaces

3.5 Additional operations

An additional operation is given by a signature fixing its name, its parameters and its result type and by an expression defining its semantics. We translate an additional operation by a transformation unit, similarly to pre-defined operations.

In Figure 7, a sample additional operation is defined for classifiers computing and returning a set of all model elements contained in the classifier, together with the contents inherited from its parents. In the espression two cases are distinguished: if there is still a parent of the classifier currently considered, the expression determines the sequential composition of three rule applications and a recursive call of transformation unit `allContents`; otherwise, nothing else is done after `contents`.

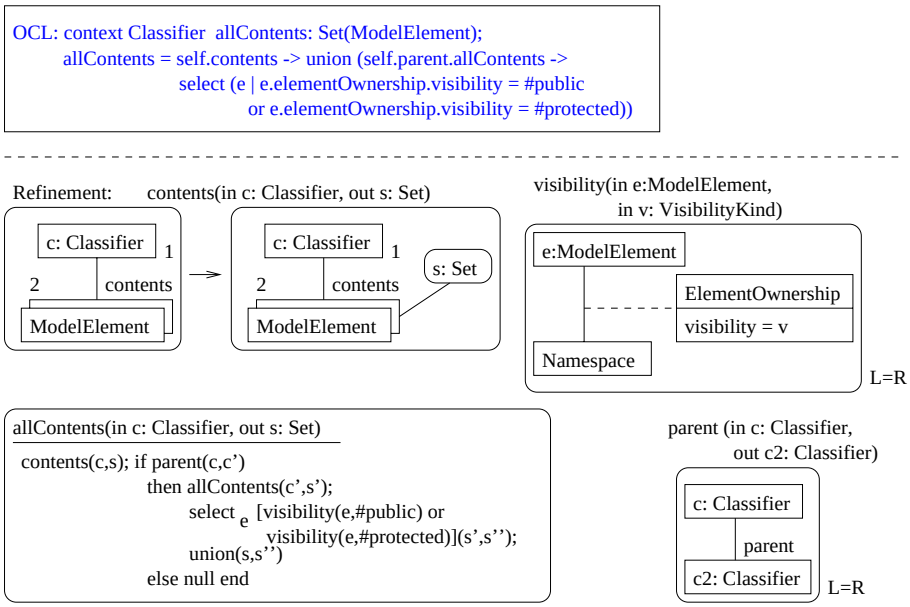

Fig. 7. Translation of additional operation `allContents`

3.6 Deletion of additional checking structure

During a constraint check, the instance graph may be augmented by additional objects and links. These objects, which may be collection nodes, are depicted by rounded rectangles. This special layout points to the fact that a certain internal attribute of the object node is set (not existing in the corresponding UML model). An object node depicted by a normal rectangle in the rule can always match a special object node in the host graph. Furthermore, additional `done` edges may have been created indicating links between collection objects and their elements. All these nodes and edges created during the checking process are deleted by the transformation unit in Figure 8 applying rules `deleteLink` and `deleteCollection` as long as possible. This transformation unit has to be applied after each check.

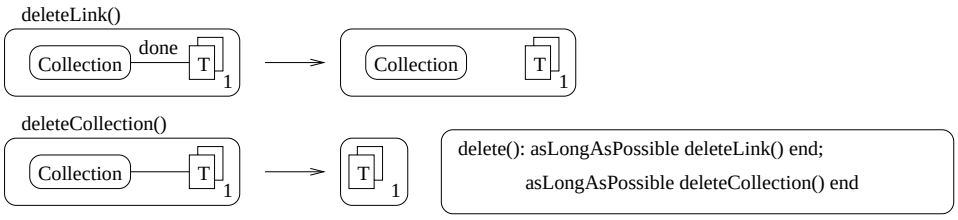

Fig. 8. Transformation unit for deleting additional object structure after checking

3.7 Consistency checking of instance diagrams

If a rule or transformation unit which is the translation of a constraint can be fully applied to an instance graph, the constraint is satisfied for this instance. For a sample consistency check, consider the simplified UML model in Fig. 9. We want to check the OCL constraint in Fig. 5 in the context of the two associations of this model.

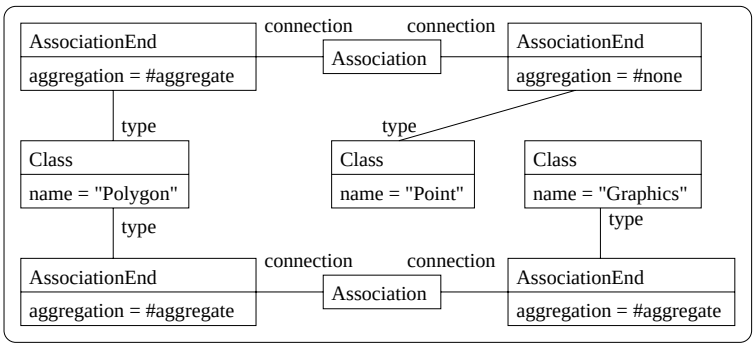

Fig. 9. A (wrong) UML model as instance of a simplified meta model

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. If a rule or unit cannot be applied completely, it stops at a certain rule trying to match it. The non-applicability of a rule can be the effect of two kinds of cause: either there is no total mapping of the left-hand side to the instance graph or all the mappings found do not satisfy all the additional application conditions. In both cases, this can be reported iteratively to the user helping him/her to find the inconsistency. Looking at our sample model in Fig. 9, the transformation unit in Fig. 5 can be applied fully to the upper association, but not to the lower one. In this case, the last rule of the expression is not applicable, because of its attribute condition.

After the user has performed further editing steps, the checking should continue with exactly that rule where the checking process stopped taking the performed changes into account. Thus, the rule-based character of inconsistency checking could be advantageously used to perform constraint checking permanently during editing, i.e. looking for rule matches after each editing step. Looking at the example again, the user has to change the lower association so that it has only one end being an aggregation, then the checking of the constraint goes through without any further complication.

4 Visual OCL

We have shown how to translate OCL constraints into graph rules and transformation units, thus providing a precise semantics for such constraints. Besides being precise, the formalization by graph transformation shows how navigation expressions, visually denoted by (some extension of) collaboration diagrams in

UML-notation, can be integrated into logical expressions, textually denoted. This combination leads to some hybrid notation of OCL which seems to be more intuitive than the pure textual form. Individual constraints can often be rendered more visually than the general translation presented in the previous section. In the following, we discuss this kind of constraint visualization in more detail and give some examples of how this new view on constraints could lead to more simplified expressions. Furthermore, we compare it with related visualizations. The running example in this section does not stick anymore at the UML-meta model, but we show visualizations of the OCL-constraints presented in [UML99] instead. Especially, when using OCL in concrete applications, a visualization might make the understanding of the constraints easier.

4.1 Visualization of constraints

A hybrid notion of OCL constraints could rely on a slightly extended UML-metamodel containing two new classes *VConstraint* and *VCollaboration*. Consider the class diagram in Figure 10. *VConstraint* stores a visual constraint with reference to its context, a set of special collaboration diagrams, called *VCollaboration*, and an optional textual constraint. A *VCollaboration* distinguishes not only standard *ClassifierRoles* and *AssociationRoles* as they occur usually in collaboration diagrams, but also negated, added and those being both. Moreover, each *VCollaboration* can have again a set of textual constraints.

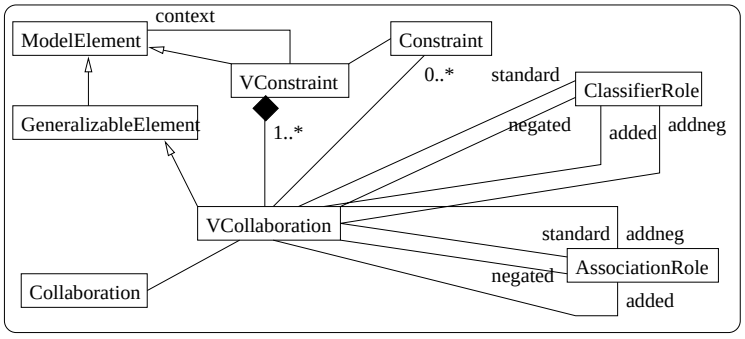

Fig. 10. The meta model for visual OCL constraints

Fig. 11 shows two visual OCL constraints, each of them with one of its possible textual translations. As done for interpretation navigation paths are described visually, whereas class attributes and their conditions are depicted textually. Depicting a paths solidly means that at least one of these paths has to exist (as in the left constraint). If part of a path is depicted dashed, the paths must not exist. In Fig. 11 the negated part is translated into a forall expression, this is possible if also the attribute condition is negated.

More complex constraints may consist of a set of subconstraints. When visualizing the complex constraint, each subconstraint is shown by a separate diagram containing a *VCollaboration*. In Fig. 12 a slightly more complex constraint

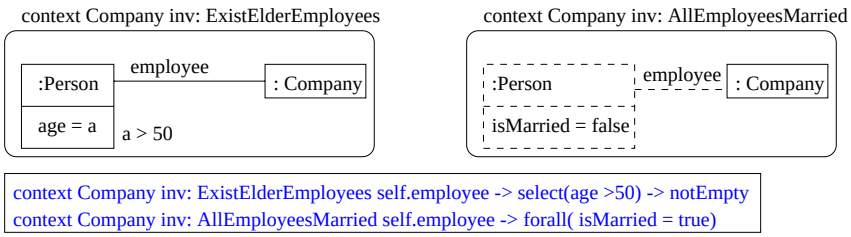

Fig. 11. Two visual OCL constraints with their textual translation

is visualized by two diagrams showing the subconstraints. The main constraint referring to these subconstraints is depicted textually on top of the diagrams. Each subconstraint describes an implication. The normally drawn diagram parts describe the preconditions whereas diagram parts depicted in bold face visualize implied conclusions. Here, an association role of sort "wife" or "husband" has to exist, then the bold face part, i.e. an attribute constraint, is checked. (Compare the textual constraint below the diagrams for these conclusions.) The extended meta model refers to implied constraints as added parts.

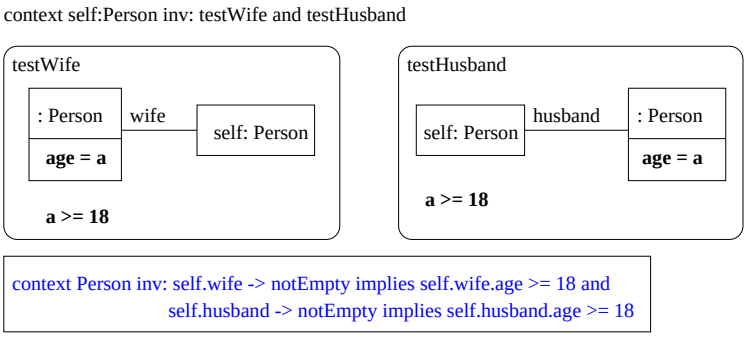

Fig. 12. A combination of visual constraints

In several cases, predefined operations on collections, e.g. *size* and *sum*, can be replaced by constraints involving only (non-)emptiness of a collection, or by applicability checks for identical graph rules. More precisely, a constraint $coll.size = 0$, with $coll$ a collection, is equivalent to $isEmpty(coll)$, and similarly for $coll.size \neq 0$. A constraint $coll.size = n$ can be expressed by an identical rule having exactly n instance elements connected to $coll$ and a negative application condition presenting one additional element. If n is too big or the constraint on predefined operations is too complex, one might prefer the textual form which is also possible to keep.

VCollaborations can be easily interpreted as identical graph rules where the graphs consist of classifier and association roles. (But unlike Section 3 we skip the annotation L=R at the graphs here.) These can come up with application conditions being a kind of implication. *VCollaborations* are combined by textual constraints translated into rule expressions as shown in the previous section. In this way, the proposed visualization of constraints comes up with a precise semantics based on graph transformation.

4.2 Related work

Kent *et al.* have studied the problem of defining a visual formalism for static expression of constraints [Ken97,GHK99], proposing first *constraint diagrams* and then *spider diagrams*, based on Euler and Venn-Pierce diagrams which allow a compact and precise set-theoretic semantics of an OCL fragment [GHK99]. For instance, it is not possible to express constraints on a region size [KH99]. A mixed formalism has thus been proposed which annotates spider diagrams with textual OCL expressions [KH99]. Spider diagrams can also be used to reason about properties of diagrams, via suitable transformation rules [HMTK99]. Spider diagrams appear to be a promising approach to the definition of a visual formalism for expressing constraints. However, our transformational approach appears to be more directly implementable with a view to automatic checking of constraints. This is facilitated by our dealing directly on collaboration diagrams, without having to recur to specific representations, as spider diagrams do. Moreover, our use of text is restricted to the specification of control flow and relations between attributes.

Evans has proposed an approach to visual reasoning on instance diagrams [Eva98], where some simple form of diagram manipulation allows relations to be inferred between elements of the diagram, e.g. elements of different subclasses of a same abstract class. Evans' proposal does not extend to consistency checking. Evans' current set of rules can be expressed by sequences of graph transformations of the type proposed here (using injective partial mappings from the left-hand to the right-hand side and sequences of rules to move associations from a class to another). Hence, it seems possible to integrate in a single tool, based on our approach, both consistency checking and reasoning on diagrams.

5 Conclusions

The paper has demonstrated a constructive way for consistency checking of OCL constraints, with a view to providing a graph-based semantics for OCL. Moreover, the adopted approach suggests a way to define OCL constraints exploiting the UML visual syntax. Based on the proposed translation, an OCL evaluator can be implemented on top of a graph transformation machine like AGG or PROGRES(in [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 based on the meta model extension shown in fig. 10 is currently implemented for the open source CASE tool ArgoUML [ARGO].

The graph transformation-based approach to checking inconsistencies can easily support automatic repair actions, by defining suitable graph rules solving such an inconsistency 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

[ARGO] ArgoUML: The Cognitive CASE Tool, see `http://argouml.tigris.org`

[BKPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, G. Taentzer. *Consistency Checking and Visualization of OCL Constraints.* Technical Report, University of Rome, SI-2000-03 , 2000.

[CKMR99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills, *The Amsterdam Manifesto on OCL*, Technische Universität München, TUM-I9925, 1999.

[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.

[Eva98] A.S.Evans, "Reasoning with UML class diagrams", *Workshop on Industrial Strength Formal Methods, WIFT'98*, IEEE Press, 1998.

[GHK99] Y. Gil, J. Howse, S. Kent, "Formalizing Spider Diagrams", *Proc. IEEE Symp. on Visual Languages '99*, 130–137, IEEE CS Press, 1999.

[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 International Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland.* IEEE Computer Society, 1999. ISBN 0-7695-0188-5.

[HMTK99] J. Howse, F. Molina, S. Kent, J. Taylor, "Reasoning with Spider Diagrams, *Proc. IEEE Symp. on Visual Languages '99*, 138-145, IEEE CS Press, 1999.

[Ken97] S. Kent, "Constraint Diagrams: Visualizing Invariants in Object-Oriented Models", *Proc. OOPSLA '97*, 327-341, 1997.

[KH99] S. Kent and J. Howse. Mixing visual and textual constraint languages. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language, Beyond the Standard*, pages 156 – 171. Springer LNCS 1723, 1999.

[RG98] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449 - 464. Springer LNCS 1507, 1998.

[RG99] M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language, Beyond the Standard*, pages 156 – 171. Springer LNCS 1723, 1999.

[Roz97] G. Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations. World Scientific, 1997.

[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.* World Scientific, 1999.

[UML99] Rational Software Cooperation. Unified Modeling Language – version 1.3. Available at `http://www.rational.com`, 1999.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.