Two Visualizations of OCL: A Comparison

Andrew Fish[1], John Howse[1], Gabriele Taentzer[2] and Jessica Winkelmann[2]

[1] Visual Modelling Group
University of Brighton
Brighton, UK
[2] Computer Science Department
Technical University of Berlin
Berlin, Germany

Abstract. We compare two visualizations of OCL, *VisualOCL* and *Constraint Diagrams*, and establish some of their benefits and weaknesses. These two notations were designed to fit in to the diagrammatic modelling paradigm. We introduce a simple case study, with constraints written in both natural language and OCL, and visualize these constraints using VisualOCL and Constraint Diagrams. Using a set of criteria which is based on cognitive, syntactic and semantic questions, we compare the notations, with reference to the sample constraints.

Keywords: Visual Formalisms, Constraint Languages, Diagrammatic Representations.

1 Introduction

Although UML [OMG04] is accepted as a standard modelling language for designing and documenting software and systems, there are still wide differences as to the use of its sublanguages. In particular, OCL [OMG03] 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 the diagrammatic paradigm. Two diagrammatic languages for expressing logical constraints in object-oriented modelling have been developed: Constraint Diagrams and VisualOCL.

VisualOCL was developed in [BKPPT01,KTW02] as a visualization of OCL and is meant as an alternative to the textual OCL. VisualOCL follows the UML notation as far as possible. This makes a direct integration of OCL in UML easier. Like OCL, VisualOCL is a formal, typed and object-oriented language.

Constraint Diagrams were introduced in [Ken97] as a visual notation for expressing logical constraints in an object-oriented system. They may be used as a modelling notation in their own right [HS05] and are independent of the UML and OCL. However, they can be used in the context of the UML as an alternative to (or a visualization of) the OCL. The basic diagrams have been fully formalized [FFH05], but a complete modelling framework involving the notation is still under development.

This comparison began with a small competition between the developers of Constraint Diagrams and VisualOCL. Each group selected a number of sample OCL constraints which were either nicely visualizable (in their opinion) using their own approach or demonstrated interesting properties of the notations. These constraints were exchanged and visualized by the other group. The most significant examples (which led to interesting statements of comparisons) are presented as OCL constraints as part of a case study in section 2.

A visualization of these constraints using Constraint Diagrams and in VisualOCL are presented in sections 3 and 4. These sections also serve for an introduction into both OCL visualizations. We ensured semantic equivalence ("informational equivalence" [LS87]) of the constraints by starting with a natural language description and translating that into an OCL expression and then into both visualizations.

The visualizations are compared in section 5. In order to conduct the comparison we present a collection of criteria which reflect different cognitive dimensions as well as other criteria obtained from cognitive science [BG00,CLS01,LS87,Shi04].

A summary of the main benefits and weaknesses of the two notations that have been observed, together with possible improvements for both of the notations is given in section 6.

2 Running example

This section introduces some simple examples of OCL constraints. OCL is a textual language which is used to describe additional constraints about objects in a UML model. The class diagram in Fig. 1 provides the context for the examples. Each example is presented in natural language and in OCL.

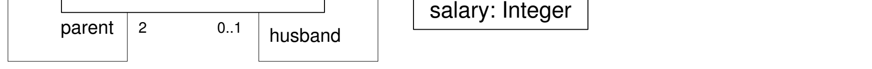


Fig. 1. Class Diagram Example

1. Each person has a different first name from last name:
```
context Person inv differentNames: self.firstname <> self.lastname
```

2. Unemployed people drive small cars and employed people drive big cars:
```
context Person def:
let carSize : Real = self.car.engine.maxPower in
if (self.isUnemployed = true )
then carSize < 1.0 else carSize >= 1.0
```

3. There is a company with no female married employees:
```
context Company inv: allInstances()->exists(c |
c.employee->select(p | p.isFemale and p.isMarried)->isEmpty())
```

4. Married men have a wife and no husband:
```
context Person inv MarriedMen:
(self.isMarried=true and self.isMale=true) implies
(self.husband->isEmpty() and self.wife->notEmpty() and
self.wife.isMale=false and self.wife.isMarried=true)
```

5. There is a company whose manager isn't allowed to work for anyone else, but may have more than one job for the company. Furthermore, all jobs of the company not held by the manager are poorly paid:
```
context Company inv: allInstances()->exists(c |
c.job->includesAll(c.manager.job) and c.manager.job->notEmpty()
and c.job->diff(c.manager.job)->forall(j:Job |j.salary<10000))
```

3 Constraint Diagrams

We present a short introduction to Constraint Diagrams using the constraints presented in section 2. Some of the constraints are not translated literally from the OCL, but expressed in a different (but semantically equivalent) manner, which either seems more natural in the notation, or it demonstrates certain properties of the notation (such as facilities for conjoining diagrams).

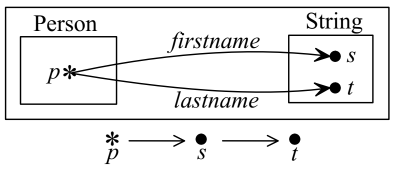


Fig. 2. Invariant 1: Each person has a different first name from last name.

The class Person is modelled by a (rectangular) contour (representing a set). We have adopted the convention to use rectangles for classes, although this is not essential.

The attributes `firstname` and `lastname` are modelled using arrows (representing relations) in Fig. 2. The asterisk labelled p corresponds to "for all p:`Person`". Quantification here is explicit, and there is no notion of *self* (although this could easily be added if one wished; for example one could write *self* instead of p in Fig. 2).

The two dots labelled s and t denote the existence of two distinct elements. The reading tree beneath the outer box provides the order in which to read the quantifiers in the diagram. Thus the diagram is read as "every person has a `firstname` and a different `lastname`".

The reading tree (below the constraint diagram) provides a partial ordering on the quantifiers (p, s, t in Fig. 2) in the diagram, thus giving a unique interpretation. Note that these trees could be replaced by an alternative method for ordering, such as using the numbering convention in the UML's collaboration diagrams. In Fig. 2 we include explicitly the information that s and t are objects of class `String`; this is a convention adopted in this paper for constraint diagrams – the rectangle labelled *String* could be omitted from the diagram as this information is available from the class diagram.

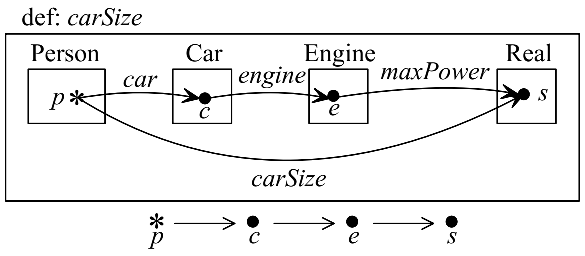


Fig. 3. Definition: *carsize*

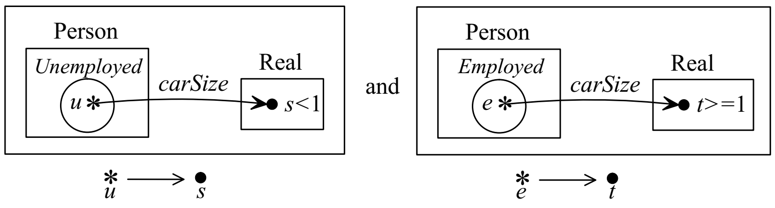


Fig. 4. Inv 2: Unemployed people drive small cars and employed people drive big cars

Navigation along associations is also modelled using arrows (again representing relations) as in Fig. 3. Thus arrows are used precisely when the *dot* notation is used in the OCL. Fig. 3 demonstrates a definition: *carSize* is defined by equating the two navigation expressions sourced on p.

The "if A then B else C" expression in Invariant 2 is equivalent to "if A then B" and "if not A then C". This alternative representation enables us to demonstrate that one can conjoin two diagrams (see Fig. 4).

States and boolean attributes are represented as sets in Constraint Diagrams. Thus in Fig. 4 the contour *Unemployed* represents the set of persons satisfying `isUnemployed = true`; similarly, the contour *Employed* represents the set of persons satisfying `isUnemployed = false`.

Labels can include simple arithmetic information (this is another convention adopted here for ease of use). For example, in Fig. 4, the label s is annotated with < 1 to indicate that the *carSize* of an unemployed person is less than one.

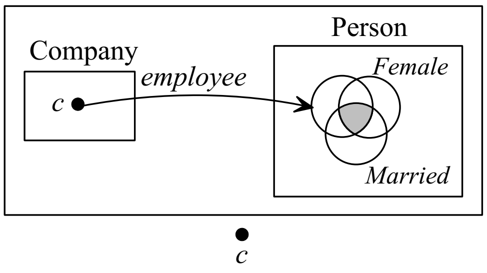


Fig. 5. Invariant 3: There is a company with no female married employees

The contours labelled *Female* and *Married* in Fig. 5 represent states and the unlabelled contour represents the set of employees of the company c. The shading in this diagram indicates that there are no elements in the intersection of these three sets – thus there are no married women employed by this company.

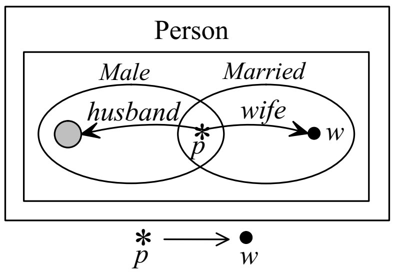


Fig. 6. Invariant 4: Married men have a wife and no husbands

In Fig. 6, *p:Person* is quantified over the set of all married men. The shading indicates that p has no husband. The arrow labelled *wife* indicates that p has exactly one wife who is not male (since the married woman, w, is the whole of the relational image of the married man, p, under the relation *wife*).

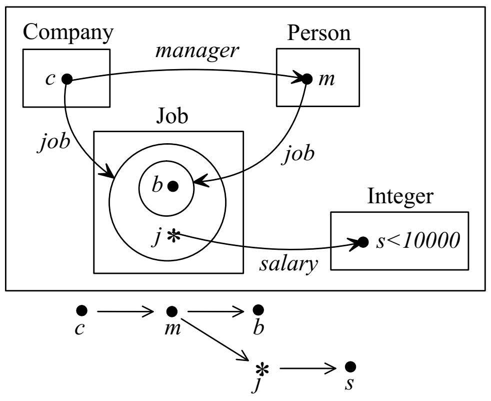


Fig. 7. Invariant 5: Hard-up company

Fig. 7 demonstrates that complex invariants can be built quite neatly. The set *c.manager.job* is contained in the set *c.job*, indicating that all of company *c*'s manager's jobs are jobs in *c*. The branching in the reading tree determines the scope of the quantifiers. The dot *b* indicates that the manager has at least one job; while *j* is a universal quantifier whose navigation expression indicates that all jobs, except the managers, have a salary of less than 10000.

The core notation has been formalized [FFH05], but a complete modelling framework involving the notation is still under development. Thus many of the notions in OCL (such as explicit notation for bags, sequences, tuples and operation pre- and post-conditions) have not yet been formalized in Constraint Diagrams (although many of these notions can be modelled within Constraint Diagrams). For example, sequences and bags can already be modelled in the (formal) core notation using relations, but shortcuts such as labelling a contour Bag(X) to indicate a bag and not a set have not been fixed yet. Constraint Diagrams can be used as a modelling notation in their own right [HS05] (where a suggested convention for pre- and post- conditions was adopted) and were developed independently of the UML.

4 VisualOCL

In this section a short introduction to VisualOCL is given using the examples presented in section 2. VisualOCL is a visualization of the whole of OCL 2.0 version 1.5. The present version 1.6 contains some new concepts, such as derived or body constraints that have not yet been integrated in VisualOCL.

VisualOCL follows the UML notation and its graphical representation as far as possible. VisualOCL is a formal, typed and object-oriented language. New data types and operations such as collections and operations like *forall*, *select* and *union* are represented by simple but meaningful graphics. Logical expressions are denoted as Peircian graphs using nested boxes to express disjunctions and conjunctions.

An OCL constraint is visualized as a rounded rectangle with two sections, the context section and the body section. The context section contains the keyword *context* followed by the type name of the model element (mostly a class or method) of the constraint followed by the kind of the constraint e.g. *inv*, *pre*, *post* or *def* (see Fig. 8 for example). Thus, the context is specified as in OCL. In the body section the body of the constraint is visualized. If the body is a navigation expression, it may have a condition section which contains textual sub conditions of the constraint declared, using variables defined in the body. If there is a condition section, it is separated from the rest of the body by a dashed line (see Fig. 9 for example).

The variable *self* is used like in OCL and is always an instance of the type of the context. If available, the navigation starts at this instance. The visualization of an object corresponds to that in UML collaboration diagrams. The attribute value of an object can be referred to by a variable. This is useful if it is to be compared with other attribute values, for example.

In Fig. 8 the variable *y* refers to the firstname of a person. The lastname is compared with the firstname. They have to be different for each person.

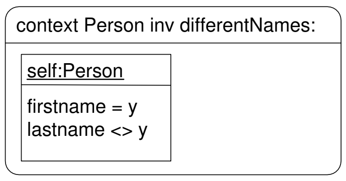


Fig. 8. Invariant 1: Each person has a different first name from last name

Object navigation is also visualized as in UML collaboration diagrams. For navigation, the role name of the opposite association end is used. In the case of unambiguous navigation, the role name can be left out. This is exactly the case if there exists only one association between the classes. The navigation result is the set of objects at the opposite end of a link and its multiplicity is defined by the corresponding association in the class diagram. The navigation on any associations always starts at object *self* if it exists, or at the context object defined otherwise. If there is only one object of the context type, this can also be used as starting point. Navigations can end at objects, association classes, attribute values, and method or operation calls.

Above we described navigation expressions, now we continue with several other kinds of VisualOCL expressions. A *let expression* defines a variable which can be used in a constraint after its definition. It is depicted by two frames, a *let* frame and an *in* frame (see Fig. 9 for example). The *let* frame contains the visualized definition of the variables. There is a separate frame for each variable where the name of the variable is depicted in the upper left corner and below the definition of the variable value follows. Inside the *in* frame a normal constraint is described which uses the variables defined above. A *let expression* is only known in the constraint in which it was defined.

The *If-Then-Else* frame contains three sections (see Fig. 9 for example). While the *If* section has to be a boolean expression, each of the other two sections can contain any VisualOCL expression. Two objects in different VisualOCL expressions are identical if they have the same name.

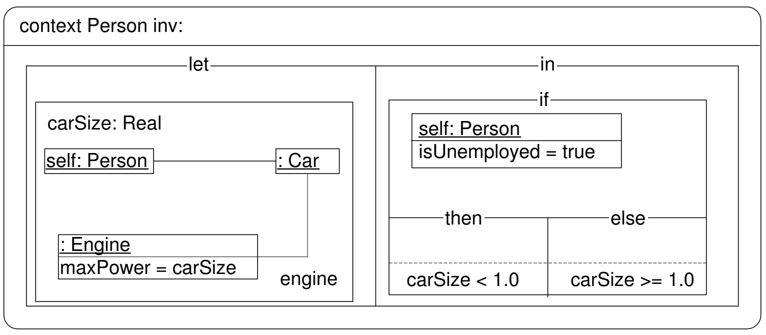
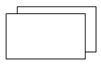
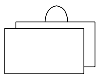
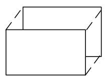


Fig. 9. Inv 2: Unemployed people drive small cars and employed people drive big cars

Collections, like sets, bags and sequences, are predefined types in OCL. The collection type is an abstract type with three concrete subtypes: *Set*, *Bag* and *Sequence*. These types are visualized as follows:

Set:  Bag:  Sequence: 

Collections have a large number of predefined operations, simple operations such as *isEmpty* and *notEmpty*, a large variety of set operations like *includes* and *diff*, or iterator operations like *select*, *forall* and *exists*. Simple operations are directly annotated at the collections. The visualization of set and iterator operations is more complex. For example, operation *exists* checks if a constraint is satisfied for at least one element in a collection. It has one iterator and an *exists* frame in which the property of the *exists* operation is visualized. The *exists* operation returns a Boolean value. On the right of the frame the ∃-operator and the iterator are depicted (see Fig. 10). Operation *select* specifies a subset of a collection. In Fig. 10 a shortcut of a *select* operation is shown. If the value over which is iterated is just an attribute of a collection element, then the shortcut for iterator operations can be used. On the right of the collection representation, the operator name is depicted. An unlabelled arrow targets the resulting collection.

An *implies* expression is visualized in an *implies* frame. Anything above or left of the keyword *implies* describes the premise. When this premise is true, it implies the conclusion denoted right or below of *implies*. Both sections may

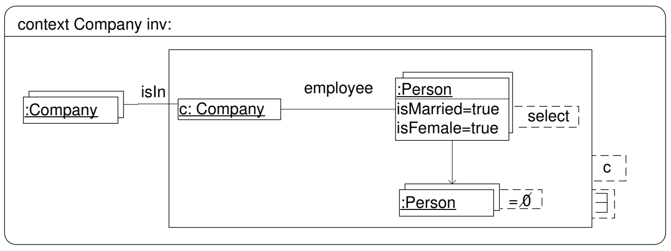


Fig. 10. Invariant 3: There is a company with no female married employees

contain any boolean expression. Note that the conclusion part of the implies operation in Fig. 11 contains a navigation expression where three subexpressions are combined by *and*.

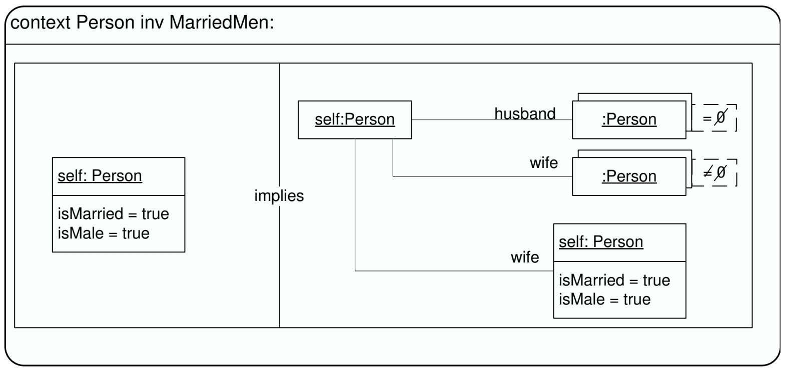


Fig. 11. Invariant 4: Married man have a wife and no husband

The difference between two sets results in a new set. This is represented by an arrow labelled by *diff(set2)* from the origin set to the resulting set. The expression *includesAll(collection2)* returns true if all elements of *collection2* are elements of the collection. Operation *forall* checks an expression for all elements of a collection and returns a boolean. In Fig. 12 a shortcut of a forall operation is shown. On the right of the collection, the ∀-operator is depicted. The navigation to association classes is again visualized as in UML collaboration diagrams.

A constraint can be composed from previously defined constraints. The constraints inside the frame are referred to by their names, thus the constraints used have to be defined first. In Fig. 13 invariants 1 and 4 are combined by *and*. Constraints can be combined by any logical operator described earlier.

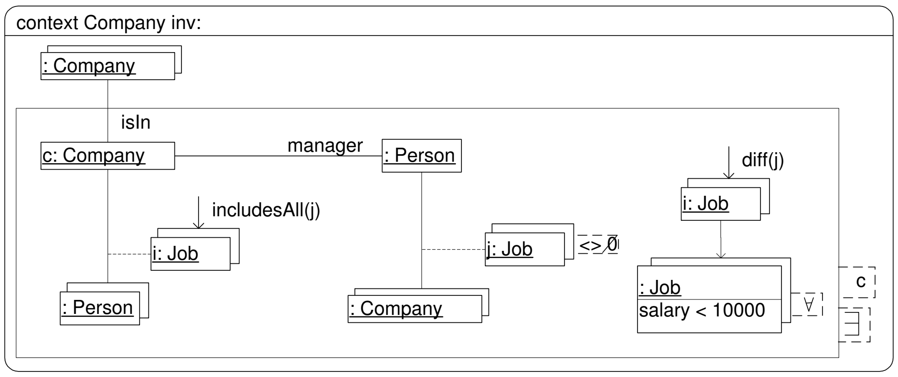


Fig. 12. Invariant 5: Hard-up Company

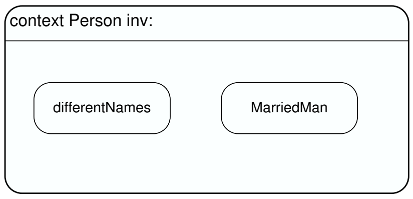


Fig. 13. Combining Constraints

5 Comparison of visualizations

We informally compare our visualizations by using criteria which reflect various notions from cognitive science [CLS01,LS87,Shi04] including cognitive dimensions [BG00].

5.1 Criteria for comparison

We will compare the two visualizations using the following list of criteria below. Structural comparisons (of logical concepts and object-oriented concepts) will occur whilst addressing these criteria.

Expressiveness: How are the syntax and semantics defined and what is the expressiveness (the *informational content* [LS87]) of the visualizations?

Effectiveness of expression:

1. *Diagrammaticity:* Where do the notations lie on the continuum between sentential and diagrammatic representations?
2. *Compactness*: How much information can be represented in a small space? Does this compactness lead to forcing the expression of extra pieces of information which do not semantically follow from the specified constraint (*specificity* [Shi04])?

3. *Ease of expression:* What sorts of expression are particularly difficult or easy to understand?

Information Retrieval

1. *Reading:* How does one read (or interpret) the diagram?
2. *Reasoning:* How easy is it to deduce (non-trivial) consequential information from the diagrams?
3. *Combining:* How easy is it to combine information from different places? This is related to *meaning derivation properties* [Shi04] – the capacity to express semantic content not defined in the basic semantic conventions, but derivable from them.

Learning Barrier: How much will a user have to learn and to be able to use the notation?

Tools: What tool support is currently available and what facilities do they have?

5.2 The comparison

Expressiveness: The formalization of Constraint Diagrams is still underway. The core notation has been given formal semantics (defined via an explicit mapping to predicate logic in [FFH05]). The general framework (including the set up for pre- and post-conditions) has not been formalised yet, although suggestions have been published [HS05]. There has been no attempt, yet, to integrate some of the concepts, such as messages, involved in OCL.

VisualOCL is a visualization of the whole OCL 2.0 version 1.5. Its syntax is precisely defined either by metamodels or by graph transformations [Win05]. The metamodel of VisualOCL is closely related to that of OCL, thus any semantics for OCL can be easily adapted to VisualOCL. At present, version 1.6 is available and contains some new concepts such as derived or body constraints. It would be very easy to extend VisualOCL to this version.

Since the Constraint Diagram notation does not yet have the syntax built in to deal with some concepts of the OCL, we cannot precisely compare the informational content of the notations.

Effectiveness of expression:

1. *Diagrammaticity*:
Constraint Diagrams have been defined to be as diagrammatic as possible, keeping text minimal. VisualOCL representations contain more text than Constraint Diagrams, since they need extra text for the context declaration, collection operations and attributes. For example, in the Constraint Diagram in Fig. 4, s and t label distinct diagrammatic elements and therefore represent distinct objects, whereas in the VisualOCL constraint in Fig. 8, the attributes are shown to be different by reference to textual labels and the <> symbol. Thus Constraint Diagrams are more diagrammatic than VisualOCL.

2. *Compactness and Ease of expression*:
   Object sets and set operations (such as set intersection and difference, shown in Fig. 7) are compactly and intuitively represented using Constraint Diagrams, since they are based on Euler diagrams. Whereas in VisualOCL, sets are represented by cascading boxes and thus set operations (such as set difference in Fig. 12) are not visualized as intuitively. For example, in Fig. 12, the label *j:Job* refers to a set of jobs (and not a single job as one might expect) which is then used by the operator in *diff(j)*. Set membership is represented spatially in Constraint Diagrams (as in Fig. 5), whilst the *isIn* operator is used in VisualOCL (as in Fig. 10).

   In both notations, navigation expressions along associations are easy to follow as well as being compact (see Figs. 3, 6, 9 and 11). From these figures, one can also see that in Constraint Diagrams attributes can be accessed in the same manner as associations (see Fig. 3), whereas in VisualOCL, attribute values are even more compact because the attributes are written textually (see Figure 9).

   Quantifiers are represented in a compact, diagrammatic way in Constraint Diagrams (see Fig. 5) and in a compact textual way for VisualOCL (see Fig. 10).

   States are represented by attribute values in VisualOCL. In Constraint Diagrams, boolean attributes may be represented as sets containing the objects for which the attribute is true and another for which the attribute is false (see Fig. 5).

   Different languages have different ways of most naturally representing constraints. There are often different ways to represent the same information and in Constraint Diagrams it often seems more natural not to literally translate the constraint, but to re-represent it as a semantically equivalent OCL statement first. For example, the “implies” from Fig. 11 is made implicit in Fig. 6, and the “if–then–else” structure of Fig. 9 has been converted in Fig. 4. This *re-representation* [CLS01] of the constraints may make the diagrams more difficult to understand from the specified OCL, due to the extra work involved in the internal (or mental) representation. However, this is a separate issue from comparing the external representations (the notations on paper; see [Heg04] for relationships between internal and external representations). Thus, one should also display the alternative, semantically equivalent version of the OCL for each such re-representation of a constraint (we do not do this for space reasons), in order to compare the external representations only. Compare the diagrams in Fig. 4 with the natural language (or an alternative translation into OCL) and it becomes less difficult to comprehend. In contrast, the VisualOCL is a more literal visualisation of OCL and thus re-representation of constraints is not required in translating between OCL and VisualOCL.

Information Retrieval

1. *Reading:* A common problem in diagrammatic systems is the lack of inherent ordering of expressions. One measure for *computational efficiency* of a

representation [LS87] is the amount of searching and memory required in a problem solving task (such as inferring information from the diagram).

In order to read Constraint Diagrams, one needs to match up the labels of the quantifiers in the diagrams to those in the reading trees underneath (thus some searching is required). Similarly, matching up operators in VisualOCL (as in Fig. 10) such as *select* and *exists* require time to search and memory storage. The adjacency of the attributes reduces the cognitive load in matching up these labels in Fig. 8.

Using the information that sets may be partitioned according to their different states may be a *hard mental operation.* For example, *p:Person* in Fig. 3 may be either *u:Unemployed* or *e:Employed* in Fig. 4.

2. *Reasoning:* Another important factor in a diagram's effectiveness (in representing information) is the operations available to the user for recognizing relevant information and for drawing inferences (consequential pieces of information) from that information [LS87].

   A formal reasoning system for Constraint Diagrams is being developed [FF04]. Such a system enables a user to perform operations (such as simple editing tasks like adding and deleting syntax) and to be told if such operations are sound (give rise to semantic entailment). Thus a user can have access to powerful operations, without the need to store them in their memory [CLS01], thus reducing the mental effort required to extract information. Reasoning rules are also imaginable for VisualOCL, on the basis of graph transformation rules, but these have not been developed yet.

   On a simpler level, one may consider what can be deduced from a diagram with little cognitive effort (the free-rides of the notations [Shi04]). For Constraint Diagrams, set containment properties (such as $A \subseteq B \subseteq C \Rightarrow A \subseteq C$) are free rides, using the topological qualities of the diagrammatic representations. For both notations, conjunction within the diagrams also comes for free.
3. *Combining:* VisualOCL supports two features for combining subconstraints: As known from UML, objects can be named and same names mean identical objects which might be visualized several times in one constraint. Moreover, subconstraints can be named and referred to by name in other constraints (see Fig. 13). Combining Constraint Diagrams has been shown in Fig. 4; a similar method of naming constraints and reusing them could be adopted for them as well.

Learning Barrier: Constraint Diagrams use fewer syntactic elements than VisualOCL. Obviously, VisualOCL is much closer in nature and structure to OCL and uses UML notation as far as possible, so it should be easy to learn for those who already know OCL and UML. People who are familiar with Euler diagrams will be accustomed to the basis of Constraint Diagrams.

Expert users of a system often have access to powerful operations which enable complex deductions. This access enables them to use the system much more effectively than novice users. The reasoning rules developed for Constraint Diagrams provide access to such operations and thus remove some of the bias towards expert users being able to use the system much more effectively than novices.

Tools: Tools for Constraint Diagrams are currently under construction by the Universities of Brighton and Kent [RWD05,VMG05]. These tools will have facilities for: drawing, editing, reasoning and translating between Constraint Diagrams and other notations, such as OCL and predicate logic. For VisualOCL a visual editor was developed as an Eclipse plug-in [Vis04] which captures all of the main concepts of VisualOCL. The user can draw and edit a VisualOCL constraint and translate it into a semantically equivalent OCL constraint using the included OCL converter.

6 Conclusion

We have compared two visualizations of OCL by using them to visualize simple constraints written initially in natural language and OCL. The notations were compared using a set of criteria drawn from cognitive science. This comparison provided an understanding of some of the benefits and weaknesses of the two notations. It will be useful to take the advantages and disadvantages of these notations into account whilst improving these notations (or whilst developing new notations). This will prove especially useful for the Constraint Diagram notation, whose development is still underway.

One interesting point is that Constraint Diagrams seem to be more useful for expressing lots of information in a compact manner with the downside that some simple expressions become harder to express (the diagrams appear cluttered). This is in contrast to VisualOCL, which is closer to the "what you see is what you get" paradigm.

As a visualization of OCL, VisualOCL has an advantage over Constraint Diagrams, whereas Constraint Diagrams is a more diagrammatic alternative to the OCL. Moreover, Constraint Diagrams can be used as a modelling notation, removing the need to consult a class diagram for example.

Improvements to the Constraint Diagram notation could be made by adopting some of the conventions in place in VisualOCL, such as the logical structures (if–then–else boxes), being less strict about the need to be diagrammatic (such as allowing attributes to be written textually rather than forcing navigation) and developing more explicit notation to express the notions within OCL.

The main recommendations for improving VisualOCL are to adopt a better means of representing set operations (perhaps extending the notation to incorporate some of the facilities of Euler diagrams) and to develop reasoning mechanisms.

We hope that the reader will agree that, even before the suggested improvements are implemented, both VisualOCL and Constraint Diagrams (which both fit into the diagrammatic paradigm of the UML) are attractive alternatives to the textual OCL .

References

[BG00] A. Blackwell and T. Green. A cognitive dimensions questionnaire optimised for users. In *Proceedings of 12th Workshop on the Psychology of Programming Interest Group*, pages 137–154, 2000.

[BKPPT01] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL using Collaborations. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language*, LNCS 2185, pages 257 – 271. Springer, 2001.

[CLS01] P. Cheng, R. Lowe, and M. Scaife. Cognitive science approaches to understanding diagrammatic representations. *Artificial Intelligence Review*, 15(16):79–94, 2001.

[FF04] A. Fish and J. Flower. Investigating reasoning with constraint diagrams. In *Visual Language and Formal Methods*, ENTCS, pages 53–67, Rome, Italy, 2004. Elsevier.

[FFH05] A. Fish, J. Flower, and J. Howse. The semantics of augmented constraint diagrams. *Journal of Visual Languages and Computing*, to appear, 2005.

[Heg04] M. Hegarty. Diagrams in the mind and in the world: Relations between internal and external visualizations. In *Proceedings of 3rd International Conference, Diagrams 2004, Cambridge, UK*, LNAI 2980, pages 1–13. Springer-Verlag, 2004.

[HS05] J. Howse and S. Schuman. Precise visual modelling. *SoSym, to appear*, 2005.

[Ken97] S. Kent. Constraint diagrams: Visualizing invariants in object oriented modelling. In *Proceedings of OOPSLA97*, pages 327–341. ACM Press, October 1997.

[KTW02] C. Kiesner, G. Taentzer, and J. Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical Report 2002/23, Technical University of Berlin, 2002.

[LS87] J. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Journal of Cognitive Science*, 11:65–99, 1987.

[OMG03] OMG. OCL 2.0 specification, revision 1.6. Available from http://www.omg.org, 2003.

[OMG04] OMG. UML 2.0 specification. Available from http://www.omg.org, 2004.

[RWD05] Reasoning with Diagrams. http://www.cs.kent.ac.uk/projects/rwd/, 2005.

[Shi04] A. Shimojima. Inferential and expressive capacities of graphical representations: Survey and some generalizations. In *Proceedings of 3rd International Conference, Diagrams 2004, Cambridge, UK*, LNAI 2980, pages 18–21. Springer-Verlag, 2004.

[Vis04] VisualOCL: Editor plugin for Eclipse. http://tfs.cs.tu-berlin.de/vocl/, 2004.

[VMG05] Visual Modelling Group. http://www.it.bton.ac.uk/research/vmg, 2005.

[Win05] Jessica Winkelmann. Specification of VisualOCL: A Visualisation of the Object Constraint Language. Master's thesis, TU Berlin, 2005. (in German).