Mixing Visual and Textual Constraint Languages

Stuart Kent

Computing Laboratory, University of Kent, Canterbury, UK
sjhk@ukc.ac.uk

John Howse

School of Maths & Computing, University of Brighton, Lewes Road, Brighton, UK
John.Howse@brighton.ac.uk

Abstract. The Object Constraint Language (OCL) is a precise language for notating behavioural constraints on UML models. Constraint diagrams have been proposed as a means of notating similar constraints, but in a visual form. This paper explores the utility of these two notations for depicting constraints, and shows how they can be used effectively together. The goal of this work is to provide more intuitive and expressive languages to support the construction and presentation of rich and precise models.

1. Introduction

The object constraint language (OCL, [10]), which is part of the UML standard [7], has been put forward as a "formal language, that remains easy to read and write" for writing unambiguous constraints that can not be captured completely in UML diagrams. Constraint diagrams (CDs, [4], [3], [6]) is a notation put forward for visualising constraints over UML models.

No doubt there is a discussion to be had on the relative intuitiveness of CDs and OCL. The increasing popularity of diagrams, such as those found in UML, suggests that for some people CDs may be more intuitive; and for others they are probably not.

Certainly, CDs are less expressive than OCL, but it is a moot point whether they could or should be extended to solve this. If they are not extended, then the intuition gained by some people using a visual notation could be countered by a lack of expressiveness. If they are extended, then the simplicity and natural intuitiveness of the diagrams might be lost.

The purpose of this paper is to contribute to the development of more intuitive and expressive languages to support the construction and presentation of rich and precise models, by showing how textual and visual constraint languages, represented by OCL and CDs, respectively, could be used together for notating constraints. The intention is not to favour one above the other, but to explore the strengths and weaknesses of each and indicate how they could be harmonised to play to their strengths.

Sections 2 and 3 overview the OCL and CDs, respectively. Strengths and weaknesses of the two notations are identified, and the relative expressivity of CDs against OCL discussed. Section 4 is the substantive part of the paper. It provides a

recipe for mixing OCL and CDs so that one can play to their individual strengths. Specifically, it considers (a) using OCL to annotate CDs, (b) embedding CDs in OCL, and (c) using a combination of (a) and (b) to go back and forth between OCL and CDs.

The fact that these notations can be intertwined, suggests that they are both concrete syntaxes for a uniform set of underlying concepts. The next stage in this work will be to formally define those concepts and mappings from OCL and CDs, with the aim of supporting the development of tools to translate between the different notations, check the well-formedness and consistency of constraints, etc. Section 5, *Further Work*, outlines a meta-modelling approach to this.

2. OCL

OCL is a form of first order predicate logic adapted for use within an object-oriented setting. Expressions in OCL always appear in the context of a class, either as invariants on that class, or as pre or post conditions of operations on that class. This paper restricts itself to invariants, but the ideas are just as applicable to pre and post conditions. This paper uses the newest version of OCL [8].

To illustrate OCL we have selected some invariants which appear in [5], which develops a meta-model for a fragment of UML that encodes semantics. The part we will focus on is the relationship between models and model instances. Specifically, a model is a package which contains just *classes* and *roles* (*association ends* in the latest version of UML). An instance of a model is a *snapshot*, which in UML is depicted by an *object diagram.* A snapshot is made up of *objects* and *links* between objects, which are themselves instances of classes and roles respectively. The class diagram for this meta-model fragment is given in Fig. 1.

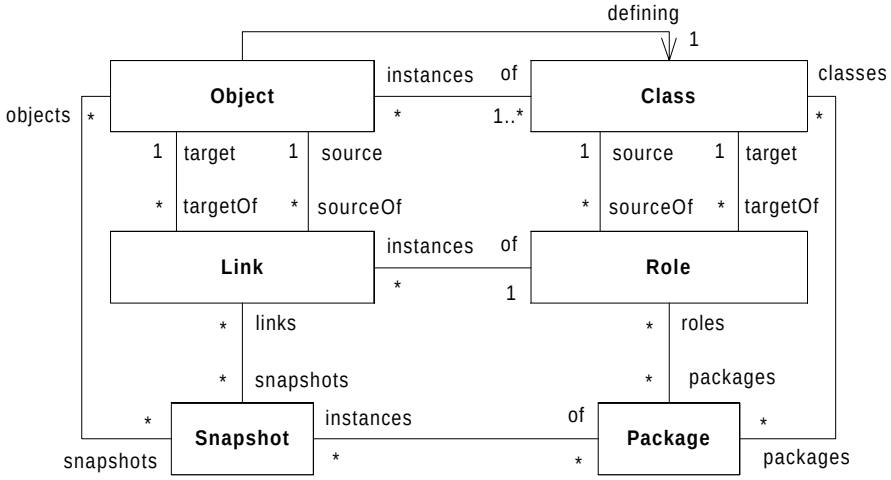


Fig. 1 Fragment of UML meta-model

Additonal constraints are required to ensure that only appropriate snapshots are instances of packages. All the required constraints may be expressed as invariants on the class `Snapshot`. The first of these is given below.

```
context s:Snapshot inv:                                          (1)
every package that s is a snapshot of, includes all the classes that s.objects
are instances of
s.of->forAll(p |
    p.classes->includesAll(s.objects.of))
```

The context part of an OCL expression indicates the context in which the expression should be interpreted. In this case, the OCL expression should be read as an invariant on the class `Snapshot`. The variable `s` is used to denote an arbitrary object of that class.

`s.of` is a navigation expression returning the set of all objects (in this case of class `Package`) connected to `s` by links of the role (or in new UML parlance *association end*) labelled `of` on the class diagram. For the object diagram in Fig. 2, representing an instance of Fig. 1, if `s` was bound to `s01` then `s.of` would return the set of objects `{p01,p02}`.

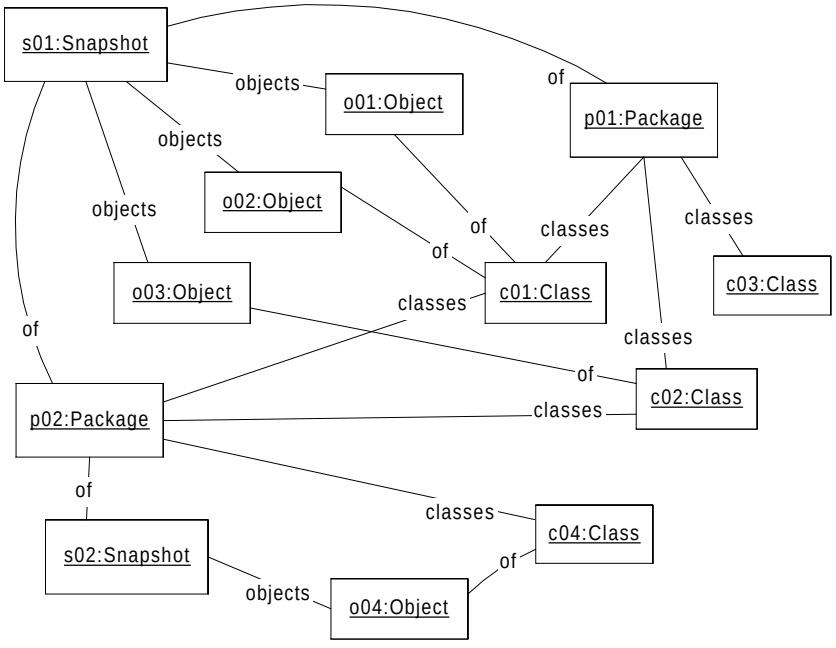


Fig. 2 Example instance of Fig. 1

`X->forAll(p | f(p))` is a boolean expression which evaluates to true if `f(p)` (which must also be a boolean expression) evaluates to true for every subsitution of `p` for an element in the set `X`. In this case `X` is `s.of`, and `f(p)` is

```
p.classes->includesAll(s.objects.of).
```

So evaluating the whole expression in the object diagram Fig. 2 for the same binding of `s` as above, requires a check that `f(p)` holds for `p` bound to `p01`, and also for `p` bound to `p02`, where these are the elements of `s.of`. The evaluation of `f(p)` for `p` substituted with `p01` proceeds as follows:

1. `p.classes` evaluates to `{c01, c02, c03}`
2. `s.objects` evaluates to `{o01, o02, o03}`
3. `s.objects.classes` evaluates to `{o01, o02, o03}.classes` which evaluates to `{c01}->union{c02}->union{c02}` which evaluates to `{c01,c02}`
4. `p.classes->includesAll(s.objects.of)` evaluates to `{c01, c02, c03}->includesAll{c01, c02}` which is true.

The evaluation of `f(p)` for `p` substituted for `p02` follows similarly.

The second constraint is very similar to 1:

```
context s:Snapshot inv:                                                    (2)
every package that s is a snapshot of, includes all the roles that s.links are instances of
s.of->forAll(p | p.roles->includesAll(s.roles.of))
```

The third constraint illustrates a number of OCL constructs:[1]

```
context s:Snapshot inv:                                                    (3)
the links of s respect cardinality constraints for their corresponding role
s.links.of->forAll(r | r.source.instances->
   intersection(s.objects)->forAll(o | let
        linksFromOForR=o.sourceOf->intersection(s.links)
            ->intersection(r.instances)
        ((linksFromOForR->size <= r.upperBound or
            r.upperbound->isEmpty)
        and linksFromOForR->size >= r.lowerBound))
    )
)
```

- `let x = exp in P(x)` is true if `P(x)` is true substituting `x` for the expression `exp`. In the constraint, the `let` expression allows `linksFromOForR` to be used instead of the much longer expression

  ```
  o.sourceOf->intersection(s.links)
       ->intersection(r.instances)
  ```

  with the obvious advantages.
- `intersection` is the complement of `union`.
- `A->size` returns the number of elements in `A`

---

[1] The corresponding constraint given in [4] is wrong - it forgets to consider that the links must be counted for each object in the snapshot of the source class of the role. This has been fixed here.

- *Numbers*. OCL includes the basic (value) types, `Integer`, `Real`, `Boolean` and `String`, in addition to collections such as sets. These are different to object types, which are UML classes.

```
context s:Snapshot inv:                                                    (4)
```
if a link is of a role with an inverse, then there is a corresponding reverse link
```
s.links->select(l1 | l1.of.inverse->notEmpty)->
     forAll(l | s.links->select(l2 | l2.source=l.target
     & l2.target=l.source & l2.of=l.of.inverse)->size=1
```

This invariant introduces a further construct:

- `A->select(a | P(a))` results in a set which is all the elements in `A` for which `P(a)` is true. In the above constraint, `select` is used to select all the links in the snapshot which are instances of roles that have an inverse.

Sets are just one kind of collection. OCL can also deal with *bags* and *sequences*. Sequences are well known, bags perhaps less so. The utility of bags often arises when summing over the result of a navigation expression that returns a collection of numbers. For example, the income of a consultant over a year is the sum of the income of each contract completed by that consultant during the year:

```
context c:Consultant,y:Year inv:                                           (5)
     c.income(y)=c.contracts->select(year=y).earned->sum
```
[2]

Now, suppose in the year there were only two contracts which happened to earn the same, say $2500 apiece. If the navigation expression returned a set of numbers, then the result would be just {2500} which, when summed, gives 2500. Clearly this is the wrong result. However, if the navigation expression returned a bag of numbers, then repeats would not be removed and the result of navigation would be {2500,2500} which, when summed, returns 5000, the correct answer. OCL has operators to coerce one collection into another. For example, `co->asSet` coerces the collection `co` into a set. If `co` was a bag, then the effect of this operation would be to remove repeated elements.

3. Constraint diagrams

Constraint diagrams (CDs) were proposed in [4] as a notation for visualising invariants, and in [6] for visualising pre and post conditions, in an object-oriented modelling context. The CD corresponding to constraint 1 in Section 2 is given in Fig. 3.

---

[2] This is not true OCL syntax, which does not allow more than one class to appear in the context. The true syntax is `context c:Consultant inv: Year.allInstances->forAll( y | ...)` We feel this is unncessarily messy.

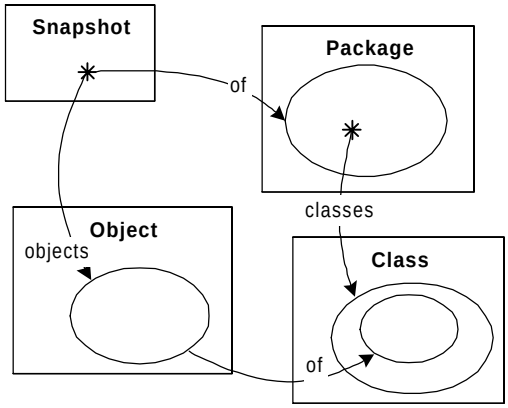


Fig. 3 CD for constraint 1

The main features of CDs are:

- *Venn diagrams/Euler circles.* CDs combine (and significantly extend) the best of Venn diagrams [9] and Euler circles [2] to show relationships between sets. In Fig. 3, the classes (represented by rectangle contours) are disjoint sets, and these each contain other sets obtained by navigating arrows (see below). In the bottom right hand corner, the set at the target of the arrow labelled `of` is contained in the set at the target of the arrow labelled `classes`. In the OCL version of constraint 1, the latter corresponds to the subexpression:

  ```
  p.classes->includesAll(s.objects.of)
  ```

- *Spiders.* Spiders are used to depict elements of sets. A spider may be a *wildcard*, representing a universally quantified variable which ranges over the region in which it is located, or *normal*, in which case it represents a specific element, which may be derived, or introduced via an existential quantifier. There are two spiders in Fig. 3, both wildcards. The first spider is contained in the class `Snapshot` and can be read as "for all elements `s` in `Snapshot`, ...". In the OCL constraint, this corresponds to: `context s:Snapshot inv: ...` ; alternatively, `Snapshot.allInstances->forAll(s |...`

  Fig. 4 shows spiders which are not wildcards. One of these spiders, z, has *legs*. Two others, `u` and `v`, are connected by a *strand*.

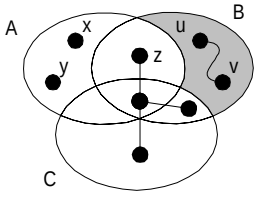


Fig. 4 Various spiders

The meaning of the diagram is as follows: there are two distinct elements, `x` and `y`, in the set `A-B-C`; there is an element, `z`, in the set

```
A->intersection(B)->union(C-A);
```

there may be one or two elements in `B-A-C`, depending on whether the spiders `u` and `v` denote the same element or not; there may not be more than two elements in this set, because shading indicates that the set is empty other than the element(s) denoted by the two spiders. So a normal spider indicates that there is an element in the set denoted by the region in which the spider is found; legs may be used to allow a spider to span more than one region; two spiders, e.g. `x` and `y`, which are not connected, denote distinct elements; two spiders, e.g. `u` and `v`, connected by a *strand*, may denote the same element or different elements; *shading* is used to show empty regions.

In this paper, we introduce for the first time the notion of a *generating spider*, which is used to generate the set of elements satisfying certain conditions. The diagrams representing constraint 4 are given in Fig. 5. The set contained in `A` is generated by the spider appearing on the contour of that set: the set denotes all the elements of `A` (which are links of some arbitrary snapshot) which are of a role that has an inverse – see below for the interpretation of arrows.

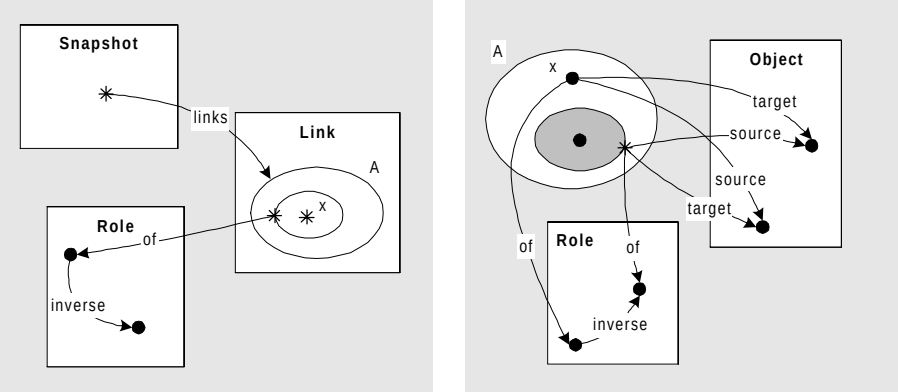


Fig. 5 CDs for constraint 4

- *Arrows*. Arrows are used to constrain relationships between sets/elements. The set or element at the target of an arrow is the result of navigating the named relationship from the element(s in the set) at the source. A CD without arrows is known as a *spider diagram* (it only contains sets and spiders).

  In Fig. 3, the set at the target of the arrow labelled `of` from the wildcard spider in the `Snapshot` box, denotes the set of elements obtained by navigating the `of` relationship (association) from the element denoted by that wildcard. This corresponds to the term `s.of` in the OCL of constraint 1. Similarly for the set at the target of the `objects` arrow. The set at the target of the `of` arrow, sourced on this latter set, denotes the set of elements obtained by taking the union of the result of navigating the `of` relationship from each element in the set at the source of the arrow (i.e. from each object in `s`). The corresponding OCL expression is `s.objects.of`

The notation is currently going through a process of improvement and refinement. The semantics of spider diagrams is in place [3], and the semantics of constraint diagrams is being written up. As explained in [3], the semantics is non-trivial for two reasons: every symbol on a CD interacts with every other symbol on the diagram; the order in which symbols are read makes a difference. The challenge has been to develop a semantics that retains one's intuitive interpretation of the diagrams.

Constraint diagrams, at least in their current form, are not as expressive as OCL. This is evident from attempts to construct the CDs for the OCL constraints introduced in Section 2. Constraint 1 was depicted in Fig. 3, and constraint 2 is very similar – no problems so far. Fig. 6 is a partial representation of constraint 3.

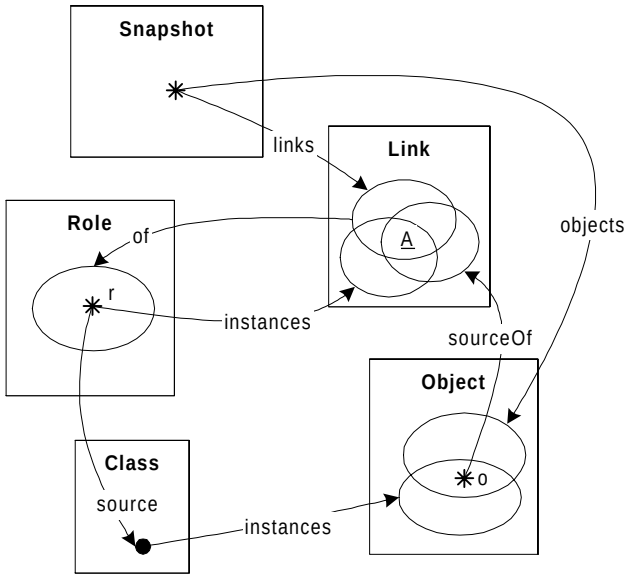


Fig. 6 CD partially representing constraint 3

The constraint diagram is very good at identifying the set of links that need to be counted: the set denoted by the region A. The problem is that there is no way of depicting the size of that set. The constraint is actually that the size of the set denoted by the region A is between `r.lowerBound` and `r.upperBound`, or not upper-bounded if the latter is empty. This could be resolved, for example by choosing a different kind of arrow to represent the OCL `->`, as opposed to the OCL `.` operator which arrows on CDs currently represent. Another approach, used in [6], is to annotate contours with numerical expressions, including variables, which are assumed to be equal to the size of the set denoted by the contour. Both these approaches add further complexity to the notation, which might make it more difficult to define and use.

The diagrams representing constraint 4 were given in Fig. 5. This illustrates the use of multiple diagrams to break down the constraint into more comprehensible chunks. The use of multiple diagrams also avoids the problem of interaction between symbols in a complex diagram. Specifically, the labels `A` and `x` are defined in the diagram on

the left; the diagram on the right then places further conditions on this set and this element. Separating the constraint into two diagrams means that we do not have to worry about the relationship between the subset of `A` introduced on the left and the subset of `A` introduced on the right[3]; nor do we need to worry about the relationships between the roles introduced on each diagram.

Fig. 7 is an attempt at depicting constraint 5. This nearly depicts the navigation routes involved in the constraint, which in the OCL version of the constraint are `c.income(y)` and `c.contracts->select(year=y).earned`, respectively. Specifically, the diagram is deficient in two respects:

1. There is no notation for bags.
2. There is no notation for `->sum`.

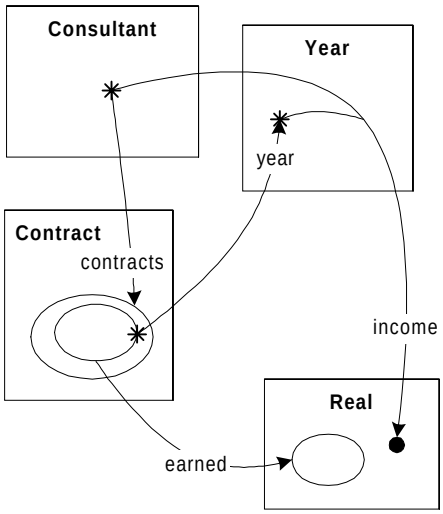


Fig. 7 CD partially representing constraint 5

A solution to (2) has already been suggested: have a special arrow for the OCL `->` operator on sets. A solution to (1) could follow along similar lines, and, therefore, suffer from the same drawbacks: introduce a different style of navigation arrow for generating bags. It is even worse in this case, because the result of this arrow will no longer be a set but a bag; but contours are interpreted as sets.

In summary, constraint diagrams seem to have much potential, and for some constraints they are ideal. However, there are other constraints, where, at some point, they are seemingly let down by a lack of expressive power. It may be possible to fix this, but at the danger of making the notation too complicated, thereby losing its simplicity and intuitiveness.

---

[3] Although, after some reflection, the reader may realise that the subset on the right will always be contained in the subset on the left if it is always the case that the inverse of the inverse of a role is the role itself.

An alternative is to harmonise OCL with constraint diagrams so that they may be freely mixed in the expression of constraints. This would allow modellers the option of using constraint diagrams where they are helpful and can be used, reverting to a textual language when this is not the case.

4. Mixing OCL with constraint diagrams

4.1. Annotating CDs with OCL

Fig. 6 visualises most of the structure of constraint 3, but not the constraint itself. It fails to work because there is no notation for depicting the size of the region A. Similarly, Fig. 7 works until the point at which the navigation expression returns a bag. The solution in both cases is simple: use labels on the diagram to provide links between the diagram and accompanying OCL expressions.

A comprehensive labelling scheme for sets and elements has already been devised for spider and constraint diagrams [3]. It works as follows:

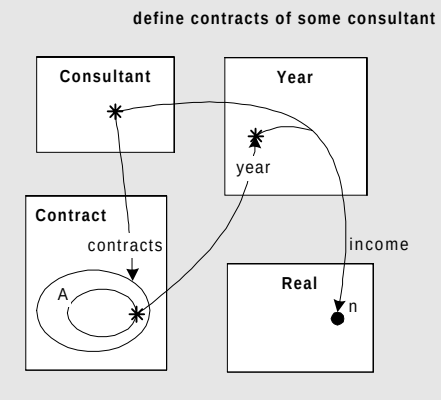
- Labels for contours are plain, with an initial uppercase letter, and are co-located with the contour.
- Labels for regions appear in the region concerned, and are underlind to distinguish them from contour labels. They also begin with an uppercase letter.
- Labels for spiders begin with a lowercase letter and are co-located with the spider.

Thus a diagram may be accompanied or *annotated* by additonal textual constraints that refer to sets and elements in the diagram. For Fig. 6, the constraint is completed with the OCL text:

```
context "Fig. 6":
    (A->size <= r.upperBound or r.upperBound->isEmpty)
        and A->size >= r.lowerBound
```

This text refers to the labels A and r in the diagram. A reference to the diagram has been used in the context section to indicate that the OCL should be read in the context of this constraint diagram.

Similarly Fig. 7 may now be replaced by a correct, annotated CD, which is given in Fig. 8.




```
context "define contracts of some consultant":
      A.earned->sum=n
```

Fig. 8 Annotated CD for constraint 5

In this case the annotation can be integrated with the diagram, by labelling an arrow with the navigation expression `earned->sum`, as in Fig. 9.

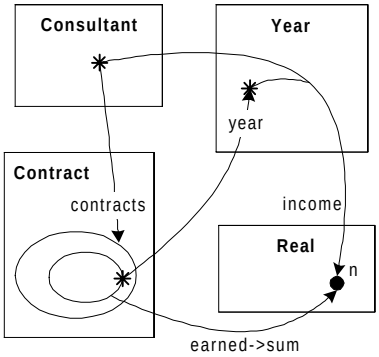


Fig. 9 CD for constraint 5 with integrated annotation

4.2. Embedding CDs in OCL

So it is possible to annotate CDs with additional OCL constraints. It is also possible to embed constraint diagrams within OCL expressions: labels are again used to connect elements and sets in the diagram to expressions in the OCL (`let` expressions in the latest version of OCL are essential here); diagrams are embedded directly in the OCL expression or referenced by e.g. a diagram name.

For example, consider constraint 6, which is a special case of constraint 3.

```
context s:Snapshot inv:                                          (6)
```
if a role with cardinality 1, then there is one link of that role per object of the source class of that role

```
s.links.of->select(r | r.upperBound=1 & r.lowerBound=1)->
     forAll(r | r.source.instances
          ->intersection(s.objects)->forAll(o |
               (o.sourceOf->intersection(s.links)
                    ->intersection(r.instances))->size=1))
```

Suppose one did not wish to use generating spiders in constraint diagrams: playing devil's advocate, one could argue that they add an extra layer of complexity to the notation, which is undesirable. Then, adopting the suggested technique, we could still use a constraint diagram to represent the main `forAll` clause of constraint 6, by embedding it in an OCL expression which provides the select clause, as in Fig. 10.

```
context s:Snapshot inv:
      let RolesOfCardinality1 =
            s.links.of->select(r | r.upperBound=1 & r.lowerBound=1)
            in
```

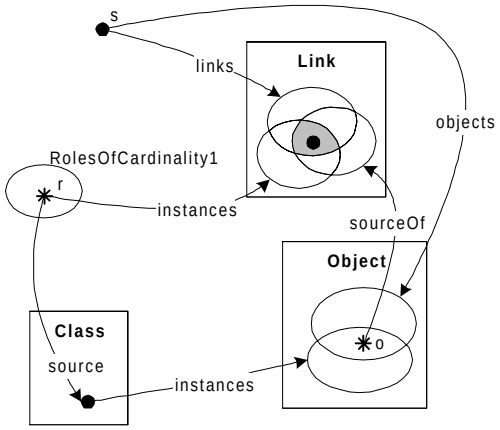


Fig. 10 Constraint 6 using an embedded CD

4.3. Mixing OCL and CDs

OCL annotations and CD embedding can be used in combination to allow arbitrary mixing of the two notations. One may begin with an OCL expression that embeds a constraint diagram which itself is annotated by OCL; or one may begin with a CD that has an OCL annotation, which embeds another CD; and so on.

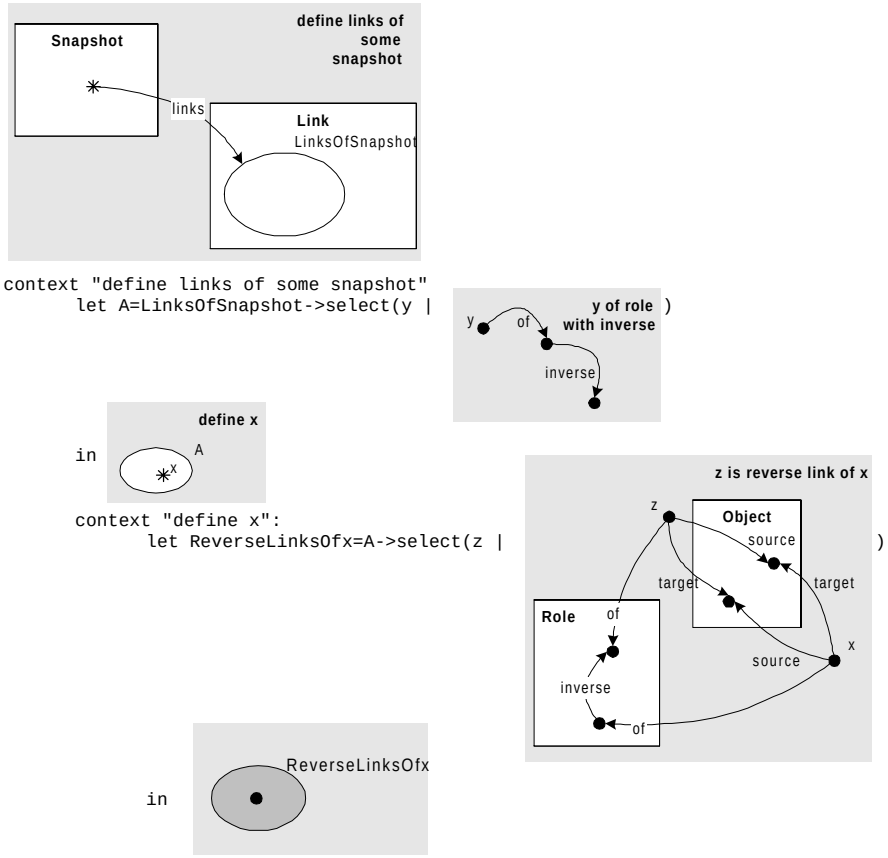


Fig. 11 Mixed representation of constraint 4

This technique is illustrated by constraint 4. Suppose, again, one did not wish to use generating spiders in constraint diagrams. Then, as illustrated by Fig. 11, constraint 4 can be represented by mixture of OCL annotations to CDs and CD embeddings in OCL.

5. Further Work

The paper has presented ways in which OCL and constraint diagrams can be used in combination to notate constraints. This suggests a promising way forward in the development of more intuitive and expressive languages to support the construction and presentation of rich and precise models. To realise this potential, a number of tasks need to be undertaken:

1. Give precise definitions of the concrete syntaxes for each notation.

2. Define a common abstract syntax that supports both notations, and define mappings from concrete to abstract, and vice-versa where possible.
3. Build tools based on 1 & 2, in particular tools that allow constraints to be entered and presented using the mixed approach.
4. Evaluate the mixed approach, versus, say, the purely textual or purely diagrammatic approach, with and without tools and to different target audiences. This does not preclude evaluation by testing the market, i.e. are the techniques widely adopted, does the tool sell?

With regard to 1 & 2 we are following the meta-modelling approach, advocated in the UML standard. Under this approach, the abstract syntax is described using a minimal core of the modelling notation itself. A fragment of the UML class diagram notation plus a constraint language, which may be any combination of (subsets of) the languages presented here, seems to be sufficient. An important difference in our approach is, perhaps, that we have managed to incorporate true semantics into the meta-model [1][5], giving rise to the possibility of tools that can check consistency of models, validate instances of models, generate instances of models (a form of partial simulation), etc. Resources are currently being harnessed to build proof of concept tools.

An enhancement of the ideas presented here, is to introduce a third notation to be mixed with the other two. Specifically, we envisage using object diagrams to show prototypical behaviour of elements. The prototypes can be embedded in the OCL or constraint diagrams: indeed there is a fragment of the constraint diagram notation which is equivalent to object diagrams (e.g. some of the embedded constraint diagrams in Fig. 11 could be replaced with object diagrams). Object diagrams could be further rendered in a domain specific visualisation: for example, an object diagram representing a communications network could be rendered in the notation for a network familiar to communications engineers.

Acknowledgements

The authors acknowledge support from the UK EPSRC under grant number GR/M02606. Thanks are also due to Steve Gaito and Niall Ross of Nortel Networks UK, and Yossi Gil of the Technion, Israel, for many valued discussions and feedback.

References

[1] Evans A. and Kent S. Core Meta-Modelling Semantics of UML: The pUML approach, in Procs. UML'99 (this volume), Springer Verlag, 1999.
[2] Euler L. Lettres a Une Princesse d'Allemagne, Vol 2, Letters No. 102-108, 1761.
[3] Gil, J., Howse J. and Kent S. Formalizing Spider Diagrams. To appear in Procs. VL'99, IEEE, 1999.
[4] Kent S. Constraint Diagrams: Visualising Invariants in Object Oriented Models. In: Proceedings of OOPSLA97, ACM Press, 1997.

[5] Kent, S., Gaito, S. and Ross, N. Putting Semantics into the UML Meta-Model. In: Kilov, H., Simmonds, I. and Rumpe, B., (Eds.) Behavioral Specifications of Business and Systems, Kluwer Academic, 1999.
[6] Kent, S. and Gil, J. Visualising Action Contracts in OO Modelling. IEE Proceedings: Software **145**, 1998.
[7] OMG UML 1.1. Specification. OMG Documents ad970802-ad970809, 1997.
[8] OMG Object Constraint Language. In: *UML Version 1.3 beta R7*, Object Management Group, 1999.
[9] Venn, J. On the Diagrammatic and Mechanical Representation of Propositions and Reasonings. *Phil.Mag.* **123**, 1880.
[10] Warmer, J. and Kleppe, A. The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998.