OCL Visualization - A Reality Check

Edward D. Willink

Willink Transformations Ltd, Reading, England,
ed_at_willink.me.uk

Abstract. Visual representations, particularly in the context of a visual paradigm such as UML, have many attractions. At OCL 2018, vOCL was presented with the well intentioned goal of making hard-to-read textual OCL more accessible within diagrams. The approach as presented was unfortunately technically unsound. This paper repairs the technical problems in the vOCL approach, and contrasts it with four other approaches.

Keywords: visual programming, OCL, QVT

1 Introduction

OCL [9] was originally part of UML, and although UML [8] is primarily a graphical representation, OCL has remained almost steadfastly textual.

Expression languages have been inherently textual for as long as mathematics has required expressions. FORTRAN set the standard for textual computer expression languages and apart from the introduction of objects and the "." dot operator not that much has changed since expression-wise. Various forms of Program Flow Diagram have been attempted to provide a graphical representation. These can be quite useful in an informal context but have generally failed to contribute to a standard language; SDL [6] is a notable exception to this observation.

Given that graphics has generally failed for expressions, it is not surprising that graphical representations of OCL have not prospered.

The novel vOCL visualization was presented at OCL 2018 despite some concerns from the submission reviewers. More concerns were expressed during the presentation. In this paper we outline these concerns and show how they can be resolved to make vOCL a genuinely novel approach to OCL visualization.

In Section 2 we present a running example and in Section 3 we present the traditional naive visualization of its OCL Abstract Syntax Tree [1]. Then in Section 4 we analyse the problems in the OCL 2018 presentation of vOCL [2] and solve them so that we can provide a vOCL visualization of our example. In Section 5, Section 6 and Section 7 we present further visualizations of our running example using QVTs [7], a Constraint Diagram [4] and Visual OCL [5] respectively. The characteristics of the five visualizations are contrasted in Section 8. Finally in Section 9 we review the related work and conclude in Section 10.

2 Running Example

Our running example uses the two-class example metamodel, shown in Fig 1. It is elaborated slightly, from that in vOCL [2], to add a Patient.id.

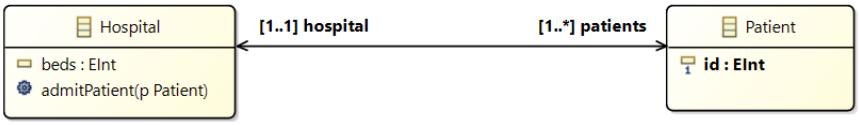

Fig. 1. Example Metamodel

The metamodel shows that a `Hospital` has a number of `beds` and may serve one or more `Patients` that have an `id`.

Our example OCL to be visualized comprises an invariant to check that each of the `patients` in a `Hospital` has a distinct `id`.

```
context Hospital
inv uniquePatientId:
self.patients->forAll(p1, p2 | p1 <> p2 implies p1.id <> p2.id)
```

(Of course, in practice, `self.patients->isUnique(id)` is simpler.)

3 Naive AST Visualization

The OCL specification[1] defines the Classes that may be used to form an Abstract Syntax Tree [1] to represent an OCL expression.

The naive AST visualization is shown in Fig 2. It is auto-generated from the XMI serialization of the AST as a UML-like[2] Object Diagram.

The diagram has been simplified to eliminate all the reference edges to, and referenced nodes for, the type/operator/property objects. It will be sufficient to study just the definition nodes. However, the diagram remains far from simple.

It is instructive to study the diagram and to reflect upon how a reader may acquire an understanding of it.

Starting at the top of the diagram, a `Class` named `Hospital` (the context) contains a `Constraint` named `uniquePatientId` as one of its `ownedInvariants`. This in turn contains an `ExpressionInOCL` as its `ownedSpecification`.

The `ExpressionInOCL` contains a `ParameterVariable` named `self` as its `ownedContext`. It also contains an `IteratorExp`, as its `ownedBody`, the root of the actual OCL expression tree.

[1] The OCL specification is imperfect. In this paper we use the similar classes from the Eclipse OCL Pivot model that prototypes solutions to OCL specification issues.

[2] Edge styling distinguishes compositions by the use bold lines with diamonds, and reference edges as dashed lines.

The `IteratorExp` has further `PropertyCallExp` and `OperatorCallExp` expressions as its `ownedSource` and `ownedBody`, and also two `IteratorVariables` named `p1` and `p2` as its `ownedIterators`.

So far, the structure is sensibly read from top down.

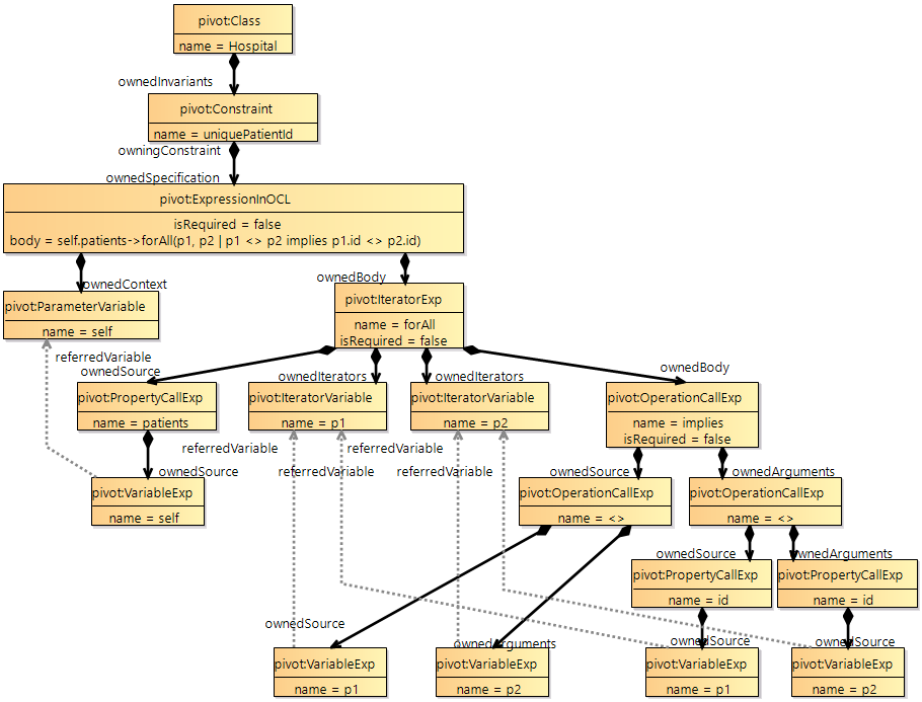

Fig. 2. Naive Visualization of Example AST

Simple expressions such as the `IteratorExp`'s `ownedSource` navigation on `PropertyCallExp` can also be read top down since we find that the navigation evaluates a `patients` property on the `ownedSource` which is a `VariableExp` named `self` referring to the overall `self ParameterVariable`. The reader can mentally push the `patients` access on to a mental stack while discovering that `self` is the source for the `patients` navigation.

More complicated expressions cannot sensibly be read top down. If we attempt to read the `IteratorExp`'s `ownedBody` top down, we encounter an `implies OperationCallExp` with a pair of `<> OperatorCallExp` expressions providing the `implies ownedSource` and `ownedArguments`. Our mental stack needs to push three of the four top down tree paths while continuing to understand one.

If instead we attempt to understand the expression part of the tree by reading bottom up, we can start at the bottom left with a `VariableExp` named `p1` that refers to the value of the `p1 IteratorVariable` and passes it as the `ownedSource` of the left hand `<> OperationCallExp`. We cannot continue up since the `OperationCallExp ownedArguments` is not yet understood. This is

resolved by yet again starting at the bottom to understand the `VariableExp` named `p2`. We now have both inputs of the left-hand `<>` and may understand that the `<>` is checking that `p1` and `p2` are different and passing this up as the `ownedSource` of the `implies OperationCallExp`. In order to fully understand the `implies` we have to go back to the bottom to understand the `p1.id` and `p2.id` navigations and `p1.id <> p2.id` comparison.

The observation that non-trivial expression trees are more easily understood by reading them bottom up is hardly surprising. An OCL evaluator can only evaluate by working bottom up, and the reader, in gaining an understanding of the OCL expression, needs to emulate that execution.

The simplified naive AST visualization requires 18 nodes and 22 edges.

4 vOCL

The vOCL [2] visualization presented at the OCL 2018 workshop introduces some interesting novelties. By re-using the Class Diagram, vOCL avoids the need for a new programming artefact, such as an OCL document or a novel UML diagram. Each OCL constraint, pre-condition or post-condition is visualized as a 'mark-up' overlay upon the pre-existing Class Diagram.

The 'mark-up' comprises a sequence of directed edges from the start point (`self`) to a sequence of nodes. A node for a `PropertyCallExp` may be created by drawing an ellipse or rounded rectangle around the role name text of the required property. A node for an `OperationCallExp` or `IteratorExp` may be created by drawing a new rectangle. The visualization is made more pleasing by the provision of a number of mnemonic icons for many common OCL operations.

Figure 2 of [2] is reproduced as Fig 3 to demonstrate this.

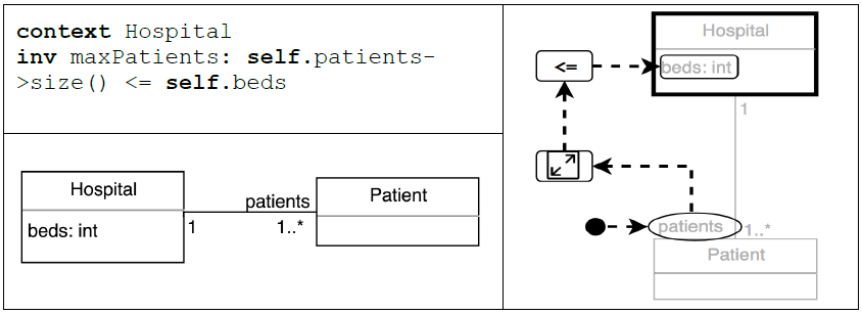

Fig. 3. vOCL's Figure 2.

At top left, we see the OCL constraint to be visualized. At bottom left, we see the Class Diagram for the metamodel. At the right we see the 'mark-up' in black upon the metamodel in grey.

The black box for `Hospital` identifies it as the context and consequently the `self` object when we start to read at the black filled circle.

The first arrow takes us from `self` to `patients` giving us `self.patients`.

The next arrow takes us to the icon for the `size()` operator giving us `self.patients.size()`.

Then on to the icon for the `<=` operator giving us `self.patients.size() <=`.

Now we hit the problem. We continue on from the `<=` to the rounded rectangle around `Hospital.beds`. Why does the result of a Boolean `<=` contribute to the evaluation of `Hospital.beds`? Where is the missing second input to `<=`? Where does the Boolean result of `<=` go to?

If we look at the naive AST visualization of this expression shown in Fig 4 and the equivalent mark-up for the reading order in green, the problem gets a little clearer.

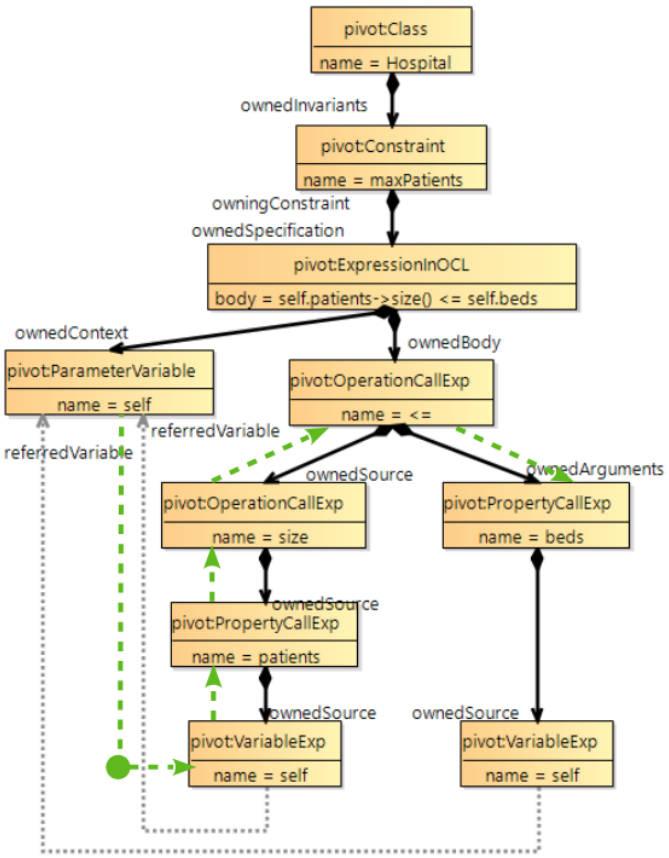

Fig. 4. Naive Visualization of vOCL invariant

The AST comprises a `<= OperationCallExp`, whose left-hand `ownedSource` is the chain of `VariableExp`, `PropertyCallExp`, `OperationCallExp` to evaluate `self.patients.size()` and the right hand side is the shorter chain for `VariableExp`, `PropertyCallExp` for `self.beds`.

It can be seen that the reading order defined by the arrows in Fig 3 take us up the left hand expression in bottom up fashion to the `<=`, but then top down on the right hand side without ever visiting the `self VariableExp`. For this simple case, intuition might perhaps suggest that `self` was implicit as the self-container of the `beds` rounded rectangle. However, in more complex cases magic rather than intuition is required to explain the semantics.

Unfortunately the vOCL paper does not really define the semantics of its reading order arrow. Rather its table provides 'Navigational arrow to follow the constraint'. If 'navigational' is meant in the modelling sense of navigation from one class via a Property to another, at most one of the four arrows in Fig 3 is 'navigational'.

When this paper was presented at OCL 2018, the intuitive nature of reading order and the inability of this representation to handle multiple instances of the same type seemed to make this approach unsound. However if we permit multiple start points, and define the reading order arrow as the data flow in the expression AST, both these problems are solved as shown in green on Fig 5.

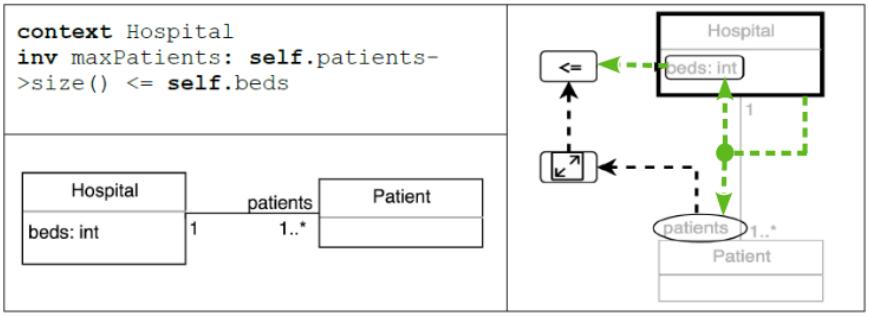

Fig. 5. Fixed vOCL's Fig 2.

The corresponding AST of this 'mark-up' is shown in Fig 6.

Once we can have multiple instances, and the reading order is clear, we can solve our example as shown in Fig 7. The three blobs correspond to the three variables `self`, `p1` and `p2`, and each contributes a partial reading.

The isolated blob for `self` is bound to `Hospital` and reads as `self.patients` to make available a `Patient` for binding to each of the other two blobs, one for `p1` and one for `p2`.

One pair of readings from `p1` and `p2`, passes via the `<>` operation to the `self` input of `implies`. Another pair of readings passes via an `id` navigation to another `<>` operation that provides the other `implies` input. The lack of an `implies` output can be interpreted as a short form for must-be-true. Note that there are two rounded rectangles around the `Patient.id` property, one for each of `p1.id` and `p2.id`.

The vOCL realization requires 9 nodes and 12 edges. The savings arise from re-use of the UML elements and from folding `Variable` and `VariableExp` nodes onto a starting blob.

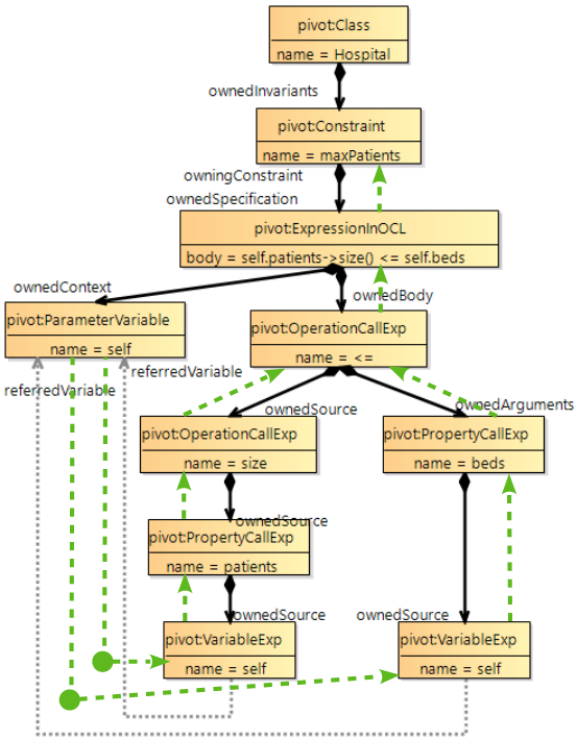

Fig. 6. Fixed Naive Visualization of vOCL invariant

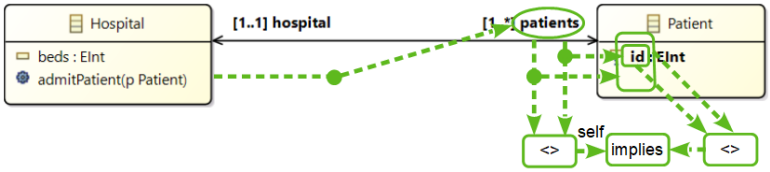

Fig. 7. Fixed vOCL for Running Example.

5 QVTs Visualization

The Eclipse QVTd project provides an alternative form of OCL visualization. This facilitates visualization of the matching and subsequent reactions of a schedulable rule. The embedding of our example in QVTr is shown below:

```
top relation isOk {
  domain from h1 : Hospital {} {
    h1.patients->forAll(p1, p2 | p1 <> p2 implies p1.id <> p2.id)
  };
  enforce domain to h2 : Hospital {};
}
```

The outer relation just transforms the `h1` instance of `Hospital` to the `h2` instance. Our example is embedded as a guard that inhibits conversion if the patient id's are not unique. The UML-like QVTs visualization is shown in Fig 8.

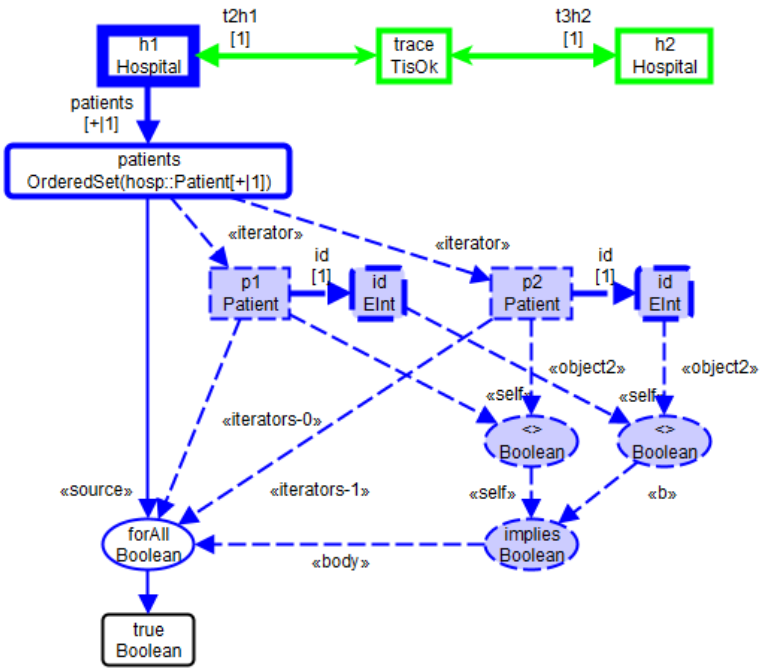

Fig. 8. QVTs Visualization of Example

The top row shows the QVT functionality matching an `h1` `Hospital` instance for conversion to an `h2` instance via a `trace` instance. Only `h1` is relevant for discussing the OCL visualization.

The thick solid line from `h1` to `patients` shows the navigation via the `patients` property from the rectangular `Hospital` Class-instance named `h1`

to the rounded-rectangular `OrderedSet` of `Patient` DataType-valued named `patients`.

The thin solid line labelled `«source»` from `patients` to the `forAll` ellipse, shows the use of `patients` as the source of a `forAll` iteration. The two iterators are shown as rectangles named `p1` and `p2`, linked by edges labelled `«iterator»` from the domain of the iteration and by edges labelled as `«iterators-0»` and `«iterators-1»` to the `forAll` to which they contribute. The remaining input of the `forAll` labelled `«body»` provides the result of the body expression of the iteration. Each of the `p1` and `p2` iterators are an input to an ellipse of a `<>` operation that feeds the `«self»` input of the `implies` operation that provides the iteration body. Each of the `p1` and `p2` iterators also provides the source to a corresponding navigation of the `id` property whose value is passed via the other `<>` operation to the `b` input of the `implies`.

Finally the thin solid line from the `forAll` iteration to the `true` literal imposes the constraint that the result of the `forAll` must be `true` for the overall QVTs rule to be permitted to execute.

Most of the elements use thin lines indicating that they are expressions whose value is checked, rather than objects whose existence forms part of the pattern match. The lines are also dashed rather than solid indicating that there may be zero or many computations to be performed. In this example the dashed lines correspond to the iteration body that may be separately evaluated for each iterator permutation. Conversely the solid line shows that the three h1-patients-forAll-true edges are always evaluated/matched exactly once.

The QVTs is a domain-specific visualization focussed on the objects/values. All OCL constructs can be visualized. The visualization is auto-generated and may only be edited to provide a more aesthetically pleasing layout.

Our example requires 11 nodes and 15 edges. The savings are mostly due to folding the VariableExp nodes that access a value into the Variable that provides the value.

6 Constraint Diagrams

Constraint Diagrams [4] are a Constraint rather than an OCL visualization. Fig 9 shows our example.

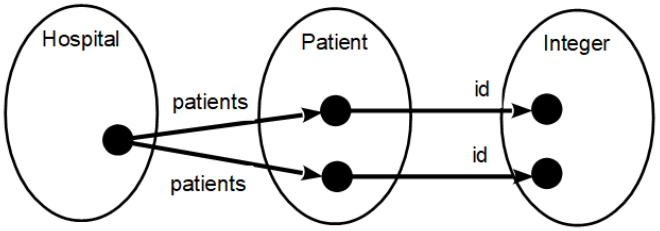

Fig. 9. Constraint Diagram Visualization of Example

The ability of the notation to depict membership of domains is well suited to this example.

The left hand ellipse depicts the domain of `Hospital` instances, with a single instance shown by the blob.

The middle ellipse depicts the domain of `Patient` instances with two distinct instances shown by two distinct blobs. Each blob is related to the `Hospital` instance by the `patients` relationship.

The right hand ellipse depicts the domain of `Integer` values with two distinct values shown by two distinct blobs. Each blob is related by the `id` relationship to a correspondingly distinct `Patient` instance.

While the notation is clearly not OCL, it is visualizing the same concepts. Blobs correspond to instances, edges to property navigations. Distinctness of the blobs realises the <> operation almost invisibly. Multiple blobs realize the dual `forAll` very elegantly. The `implies` operation is also invisible, arguably because the Constraint Diagram implements a slightly different logic; the possibility that the iterators are the same is just not possible.

This example demonstrates how Constraint Diagrams have a 'sweet spot' that can often coincide with typical usage. However when something else is required more complexity is needed. There is no support for arbitrary OCL.

The example requires 8 nodes and 4 edges. There is very little extra text.

7 Visual OCL

Visual OCL re-uses typical UML graphical idioms for its visualization, particularly the compartment nesting for hierarchical state machines. Fig 7 shows our example. (The figure is an adaptation of Fig 2.32 of [5].)

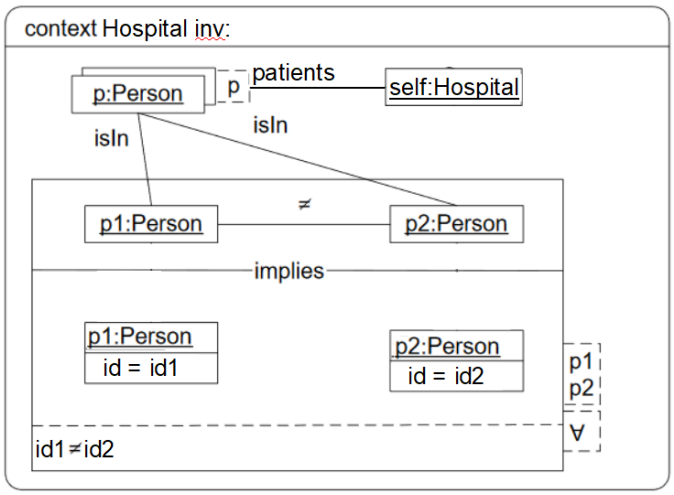

Fig. 10. Visual OCL Visualization of Example

The outer compartment is top-titled to show that it is an invariant of a `Hospital`. It contains a side-titled compartment for a ∀ with `p1` and `p2` iterators. This in turn contains a pair of compartments for left and right hand side of an `implies` operation. The lower compartment is split so that a textual expression can express what cannot be expressed graphically.

Within most of the compartments, a small UML pattern identifies a part of the expression.

In the outer compartment, a self instance of `Hospital` has a `patients` navigation to the `p` multi-instance of Patient.

The top half of the `implies` identifies two `Person` instances named `p1` and `p2` with a ≠ relationship between them, and the special `isIn` relationship to the outer `Person` multi-instance.

The bottom half of the `implies` again shows the two `p1` and `p2 Person` instances but now with identification of the `id` attributes, whose value is constrained by the ≠ in the textual box.

Given a particularly helpful UI, it is perhaps possible to draw the diagram with 12 node/compartment creations, 3 edge creations and 10 text token selections/entries.

Contrasting Visual OCL with the naive visualization suggests that the main difference is that the compositions of the naive visualizations are replaced by compartment nesting. Since both compositions and compartments are limited to a single parent, the use of nested compartments seems sound and so the only limitation is in the diversity of helpful idioms such as the ∀ iteration side-titling and a guarantee that whatever lacks a special idiom can be captured by a textual compartment.

In the example, use of ∀ and ≠ rather than `forAll` and `<>` illustrates a dilemma that has been resolved by favouring a mathematical notation rather than the OCL which is itself a compromise between compactness for mathematicians and readability for casual programmers. Not following OCL seems like a mistake since ultimately the textual compartment should be OCL rather than OCL-like.

8 Comparison

The characteristics of the five visualizations are summarized in the following table.

Visualization	Nodes	Edges	Texts	OCL Coverage	Diagram Editing
Naive AST	19	22	-	Full	Auto-generated
vOCL	9	12	-	Full	Mark-up
QVTs	11	15	-	Full	Auto-generated
Constraint Diagram	8	4	-	Partial	Manual Edit
Visual OCL	12	3	10	Full	Manual Edit

The nodes and edges indicate how many graphical artefacts are used by the visualization of our running example. For most of the visualizations, the text

is rendered automatically from metamodel elements. Only for Visual OCL does additional text appear to need manual entry.

The Constraint Diagram has distinctly lower artefact counts, but it has only partial coverage of OCL syntax. Our running example may be unfairly well suited to a Constraint Diagram visualization.

Two of our visualizations are secondary; they are auto-generated from the textual OCL or from QVTr. The other three are primary, requiring an appropriate graphical editor. vOCL requires an ability to mark up a standard UML Class Diagram. The remaining two require editing support for a new kind of Diagram.

The need for new editing capabilities is unattractive, but even the auto-generated visualizations need improved tooling to provide adequate layouts. The figures in this paper required considerable manual enhancement.

9 Related Work

This paper was motivated by a need to solve the problems in vOCL [2]. The solutions and examples are presented here in Section 4.

We identify the Dragon Book [1] as a source for the traditional naive AST visualization.

Alternative visualizations have been provided as Constraint Diagrams [4] and as Visual OCL [5]. We provide examples of these diagrams in Section 6 and Section 7.

We identify a more modern OCL visualization embedded within the QVTs visualization of the Eclipse QVTd project [10].

Overall our comparison of five alternative visualizations is as inconclusive as the two contrasted in [3].

10 Conclusions

We have remedied the deficiencies in vOCL so that its 'navigation' arrows have a consistent data flow semantic, and so that multiple start points can accommodate non-trivial expressions.

We have presented five alternative visualizations of an OCL expression of moderate complexity. It is far from clear that any of the visualizations is unambiguously better than any other. Each can be good for a particular usage.

vOCL and Constraint Diagrams may be better within a UML tool, but the attempt to hide OCL completely seems unrealistic, since OCL must be used when the expression gets too complex. The new graphical idioms therefore just add to the amount to learn and require tooling to provide new diagrams.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers, Principles, Techniques and Tools, Addison Wesley, 1986

2. Badreddin, O., Barraza, G., Hamou-Lhadj, A.: vOCL: A novel approach for UML constraints modeling, 18th International Workshop in OCL and Textual Modeling, October 14, 2018, Copenhagen, Denmark. http://ceur-ws.org/Vol-2245/ocl_paper_7.pdf
3. Fish, A., Howse, J., Taentzer, G., Winkelmann, J.: Two visualizations of OCL: a comparison. Brighton, UK, 2005. https://research.brighton.ac.uk/files/186722/VOCLTR.pdf
4. Kent, S.: Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. OOPSLA 1997. https://www.cs.kent.ac.uk/pubs/1997/794/content.pdf
5. Kiesner, C., Taentzer, G., Winkelmann, J.: Visual OCL: A Visualization of the Object Constraint Language, TU Berlin Technical Report 2002/23, 2002. http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/gKTW02.pdf
6. Rockstrom, A., Saracco, R.: SDL-CCITT Specification and Description Language,IEEE Transactions on Communications 30(6):1310 - 1318, July 1982, https://www.researchgate.net/publication/224733068_SDL-CCITT_specification_and_description_language
7. Willink, E.: QVTs: A TGG-like graphical representation for efficient Declarative Model to Model Transformation scheduling, March 2019, https://www.eclipse.org/mmt/qvt/docs/ICGT2019/GraphicalQVT.pdf
8. OMG Unified Modeling Language (OMG UML), Version 2.5, OMG Document Number: formal/15-03-01, Object Management Group (2015), http://www.omg.org/spec/UML/2.5
9. Object Constraint Language. Version 2.4., OMG Document Number: formal/2014-02-03, Object Management Group (2009), http://www.omg.org/spec/OCL/2.4
10. Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd