

vOCL: A novel approach for UML constraints modeling

¹Omar Badreddin, ²Gerardo Barraza, ³Wahab Hamou-Lhadj,
^{1,2}University of Texas, El Paso, TX, USA
¹obbadreddin@utep.edu
²gbarraza5@miners.utep.edu
³Concordia University, Montreal, QC, Canada
³wahab.hamou-lhadj@concordia.ca

Abstract. Object Constraint Language (OCL) supports UML model navigation, queries, and constraints. OCL has precise unambiguous semantics and supports definition of constraints at the model level. However, OCL is arguably the least adopted UML notation in practice. We argue that while the language constructs, concepts, and semantics are well founded, the language's surface textual notation significantly hampers comprehension. Its navigational features mean that users must track a linear textual syntax against a visual model with a 2-D layout.

This paper proposes a novel visual surface notation that maintains its foundational strengths in supporting navigation and querying, and improves on its representation and comprehension. The proposed visual surface notation is superimposed on its UML model context to enable the visualization of one or more constraints along with its related contextual model elements. Improving on constraints readability and comprehension ultimately improves constraints adoption in practice, and it enhances the learning experience in classrooms.

The contributions of this paper are a novel constraints surface notation, a supporting meta-model, and an evaluation of the proposed approach using a controlled experimentation.

1 Introduction

UML is an ecosystem of standards and tools to support model-based software and systems engineering. UML includes thirteen diagrams that address various aspects of systems development. Model Driven Architecture promotes the use of models, rather than code, as the primary development artifact. The premise includes improvements in software quality and reliability, as well as improvements in engineers' productivity. Towards that goal, a set of new standards and supporting tools are emerging that formalize model semantics, effectively reducing the abstraction gap between models and code. Action Language for Foundational UML (ALF) is a textual language with precise semantics that are based on a subset of the UML meta-model 10. ALF supports many statements commonly supported in many existing object-oriented languages, but also supports statements at higher level of abstraction that are more commonly found in models such as class diagrams and state machines.

The need for precise models is well founded. Models that lack precision are consequently ambiguous, subject to misinterpretation, and have limited role in contributing to automated code generation. Object Constraint Language (OCL), like ALF, is a textual language that defines model-level constraints. It supports model/graph navigation, and includes executable functions on collections and sets 11.

Despite OCL's capabilities, its adoption in practice is rather limited. OCL is arguably the least used UML notation in practice 12. To use OCL, engineers must trace sequential textual constraints and navigate against a visual model of the system. We hypothesize that the textual

notation of the language is a key hindrance, as well as poor tooling and weak specifications. Existing approaches to visualize such constraints are limited or ambiguous, and does not address the fundamental limitations of comprehension and usability.

The contributions of this work are a novel approach and a demonstration for OCL-based constraints visualization, a supporting UML metamodel extension, and an evaluation of comprehension value using a controlled experiment. The novel surface notation is superimposed on the corresponding UML model elements (the context diagram). We demonstrate by example how the novel notation and visualization approach improves on constraints comprehension. We also demonstrate how the proposed approach improves on existing state of the art in constraints modeling and visualization. We also introduce a meta-model of the proposed notation as an extension to the UML metamodel.

The rest of the paper is organized as follows. In Section 2 we provide a background on OCL. We discuss related works in Section 0. The proposed visual notation is presented in Section 4. The proposed language specification and meta model are presented in Section 5 and 6 respectively. We discuss the limitations of the proposed approach in Section 7. We conclude and layout future works in Section 8.

2 Background

In this section, we introduce a necessary background on OCL and its key statement classes. OCL was initially developed at IBM, and was later merged into the UML standard suite 13. OCL started as a complement of the UML notation with the goal to overcome the limitations of UML (and in general, any graphical notation) in terms of precisely specifying detailed aspects of a system design [3]. Initially, OCL's scope was limited to specifying constraints for UML Models. OCL is now a key modeling notation for many model-driven engineering (MDE) technologies, such as model transformation, validation, model-based querying and reporting.

Visual languages are generally desirable as they tend to be easier to comprehend and are typically superior as a communication medium. However, visual language designers face an engineering trade-off between language precision and the number of unique language concepts/constructs and their corresponding visual elements. To improve precision, the language must support more of such elements, making the language harder to learn, harder to understand, and ultimately undermine the primary objective of the language itself. For that reason and due to its origin and mathematical foundation, OCL is a textual language and it is designed specifically to complement diagrams with precise constraints statements [4].

OCL is declarative; it does not support imperative constructs such as assignment statements and supports the following key constructs: 1) Invariants, conditions that must be true at all times; 2) Derivation rules that define computations of model elements; 3) Model querying, or operations on collection data sets; and 4) Pre and post condition definitions. OCL supports mathematical expressions, binary operations, as well as OCL specific operations (such as `oclIsNew()`, `oclIsUndefined()`), standard operations, collection operations, and iteration operations on collections. Furthermore, OCL has four collection types: Set, OrderedSet, Bag, and Sequence. The following is a simple OCL Constraint that define an upper limit for the number of patients in a hospital.

```
context Hospital  
inv maxPatients: self.patients->size() <= self.beds
```

3 Related Work

Constraints are frequently expressed in natural language, even when system designs are expressed in UML modeling notations 14. Despite its appropriateness, OCL has not attracted any significant interests in practice 1. In fact, it has been argued that OCL is the least adopted UML standard notation 15. This has motivated researchers to investigate approaches to improve OCL comprehension and representations.

Visual OCL proposes a visual representation of various OCL textual constructs 1. Constraints are presented in the form of compartments that contain tags that correlate to elements in the UML diagram. These tags are connected to one another to demonstrate their association with one another and construct the visual constraint. Visual OCL are independent models (i.e. separate from the context) with no support for context diagram navigation. Users of Visual OCL, in addition to learning the OCL syntax, must also learn the non-trivial mapping from OCL statements to Visual OCL compartments and these associated tags.

Constraints Diagrams 1619 is another visual constraint modeling notation that uses rectangular boxes containing nodes and arrows representing relationships between elements of a class diagram can be arranged to construct a constraint nearly equivalent to an OCL constraint. Later theoretical and empirical investigations suggest significant improvements to comprehension associated with Constraints Diagrams models 21.

Fish et al 2 18 conducted a cognitive evaluation of the textual OCL notation against two visual constraints modeling approaches; Visual OCL and Constraint Diagrams. The evaluation focused on the effectiveness of the visualization using a number of characteristics, including completeness, ease of expression, information retrieval factors, reasoning, learning, and ability to combine multiple sub-constraints. While Constraints Diagrams were more expressive, they lacked flexibility and navigability of the corresponding Visual OCL models.

Bottoni et al propose a UML metamodel extension to support constraints. Their approach minimizes the introduction of new visual elements by extending the UML Collaboration Diagram metamodel 17. Their approach proposes new visual elements to represent various OCL operations (Fig. 1).

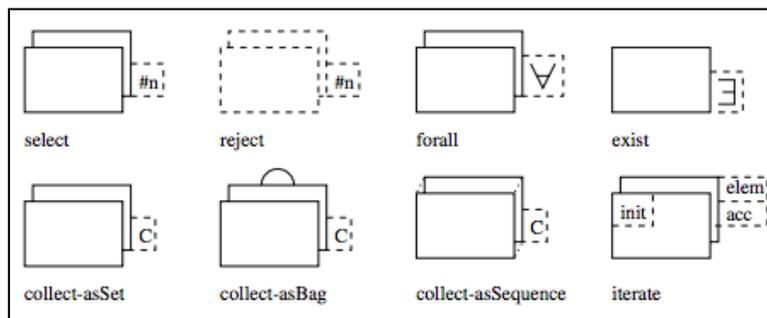


Fig. 1: A Visualization of OCL using Collaborations [17]

VMQL is another approach to visualize OCL constraints proposed by Störrle 20. Despite being less expressive than OCL, VMQL has demonstrated significant improvements in comprehension and maintainability.

4 Proposed Superimposed Visual OCL Notation

We propose vOCL, a novel visual surface notation for OCL. The key novelty in this approach is twofold. First, vOCL language is expressed as part of the context UML model, and not as a

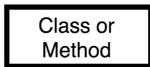
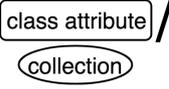
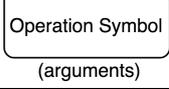
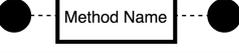
separate model entity. As such, the language supports visualizing one or more constraints on parts or on whole models. Second, the language reduces dependence on the existing OCL textual notation, and define new modeling elements to represent a foundational core of OCL constructs. The rationale for this language design is to minimize the distinction between the model and its constraints, and improve language comprehension. As such, the language is designed as extensions to the UML instance model [5]. Practical implications of this approach include facilitating the integration of the language into existing UML modeling tools.

The following section demonstrates key language constructs. Demonstration of the language design approach is demonstrated by a running example in section 5.

4.1 Key vOCL Elements

We cover three types of constraints – invariants, and pre and post conditions. The following table summarizes key vOCL statements and their corresponding OCL syntax. vOCL is adapted from OCL and thus deviations from actual OCL constructs are kept to a minimum. The language introduces a few new symbolic elements to aid model navigation, such as starting node and navigational arrows. Table 1 summarizes key vOCL constraints.

Table 1. vOCL core constraint constructs

	vOCL Elements	OCL Syntax	vOCL Symbol	Description
Basic Elements				
1	Context of constraint statement	Context: <<class name>> or <<class::method()>>		Indicates which class or method the constraint applies to – denoted in bold
2	Initial element of invariant constraint	inv (name optional): <<constraint>>		Indicates starting point of constraint statement
3	Directional flow of constraint	No OCL Syntax		Navigational arrow to follow constraint
4	Participating element in constraint statement	Attribute name/Collection name		Element that is part of constraint – could be a class attribute or a collection
5	Operations and operators	<code>select()</code> , <code>size()</code> , <code>forall()</code> , <code><=</code> , etc.		Operation that is to be applied to a participating element in the constraint
6	Pre and Post Condition	pre: <<condition>> post: <<condition>>		Pre-condition starts at left, post-condition at right
Operations				
Standard Operations				
7	Instance count	<code>count (e)</code>	1 2 3 ...	Returns count of instances of e in collection
8	Exclusion	<code>excludes (e)</code>		Returns true if e is not in the collection
9	Inclusion	<code>includes (e)</code>		Returns true if e is in the collection
10	Empty collection	<code>isEmpty ()</code>		Returns true if collection contains no elements

	vOCL Elements	OCL Syntax	vOCL Symbol	Description
11	Non-empty collection	<code>notEmpty()</code>		Returns true if collection contains at least one element
12	Summation	<code>sum()</code>		Returns the sum of all elements in the collection
13	Size of collection	<code>size()</code>		Returns the number of elements in the collection
Collection Operations				
14	Insert at end of collection	<code>append(e)</code>		Appends element e at the end of collection
15	Convert to bag	<code>asBag()</code>		Returns collection as Bag: Not Ordered, Not Unique
16	Convert to ordered set	<code>asOrderedSet()</code>		Returns collection as Ordered Set: Ordered, Unique
17	Convert to sequence	<code>asSequence()</code>		Returns collection as Sequence: Ordered, Not Unique
18	Convert to set	<code>asSet()</code>		Returns collection as Set: Not Ordered, Unique
19	Element at index i	<code>at(i)</code>		Returns the element at index i
20	New collection excluding e	<code>excluding(e)</code>		Returns new collection without any instances of e
21	New collection including e	<code>including(e)</code>		Returns new collection with an instance of e
22	First element	<code>first()</code>		Returns the first element of collection
23	Index of element e	<code>indexOf(e)</code>		Returns the index of the first appearance of an instance of e
24	Insert element at i	<code>insertAt(i, e)</code>		Returns collection with element e at index i
25	Last element	<code>last()</code>		Returns the last element of collection
26	Insert at beginning of collection	<code>prepend(e)</code>		Prepends element e at the beginning of collection
Iteration Operations				
27	Collection of elements in self	<code>collect(expr)</code>		Returns a bag of elements for which expr is true
28	Existential	<code>exists(expr)</code>		Returns true if the collection has at least one element for which expr is true
29	Universal	<code>forall(expr)</code>		Returns true if expr is true for all elements in the collection
30	Unique	<code>isUnique(expr)</code>		Returns true if expr has a unique value for each element in the collection

	vOCL Elements	OCL Syntax	vOCL Symbol	Description
31	Iterator	Iterate (i:Type;a:Type expr)		Base iteration operation
32	One element	one (expr)		Returns true if only one element in the collection satisfies the expr
33	Selection of elements	select (expr)		Returns a collection with all elements for which expr is true
34	Sorting collection	sortedBy (expr)		Returns a collection sorted according to expr

4.2 vOCL-UML Model Integration

The vOCL meta model (discussed later) is designed as a set of extensions to the UML meta-model. As any constraint statements, vOCL can only be visualized on the context diagram. In the case of vOCL, the constraints and the model are unified in a single diagram; or rather, the vOCL constraints are superimposed on the model. One or more vOCL constraints can be applied to the same UML context diagram. To demonstrate this approach, consider the following simple UML class diagram model and constraint.

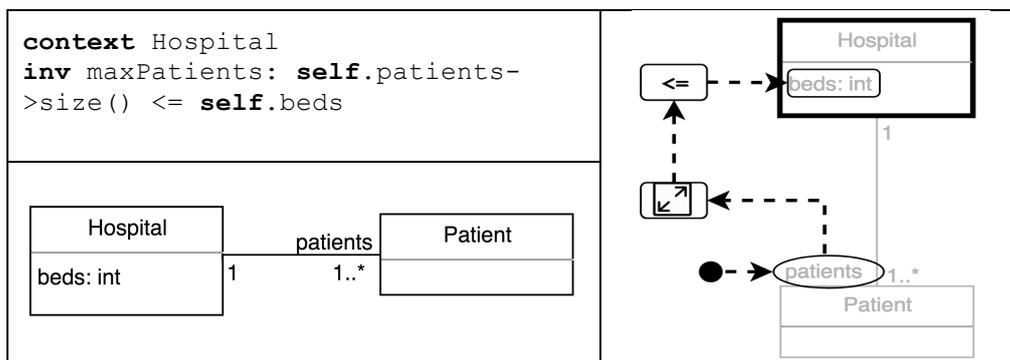


Fig. 2: Example of vOCL and UML model integration

Two classes, Hospital and Patient, of which, Hospital has a collection called “patients” of type Patient. Consider the constraint mentioned earlier where the number of patients a hospital has must be less than or equal to the number of beds. The corresponding vOCL model is demonstrated in Fig. 2. vOCL is integrated directly on top of the context diagram. The class Hospital is bolded to indicate that this is the context element – or *self* using OCL syntax.

To demonstrate pre and post conditions in vOCL, consider the OCL statement and the corresponding vOCL in Fig. 3. This OCL constraint specifies that before the execution of the `admitPatient()` method, the number of beds in the hospital should be greater than 0. This must be true in order to carry out the method. It also states that when the method terminates, the new collection of patients should be equal to the old collection of patients (thus the OCL “pre” keyword indicating the collection before execution) but will now include the new patient *p* that was added. The inclusion is denoted by the including() operation symbol which requires a new parameter– in this case, it is the value *p* – the value in the method’s parameters.

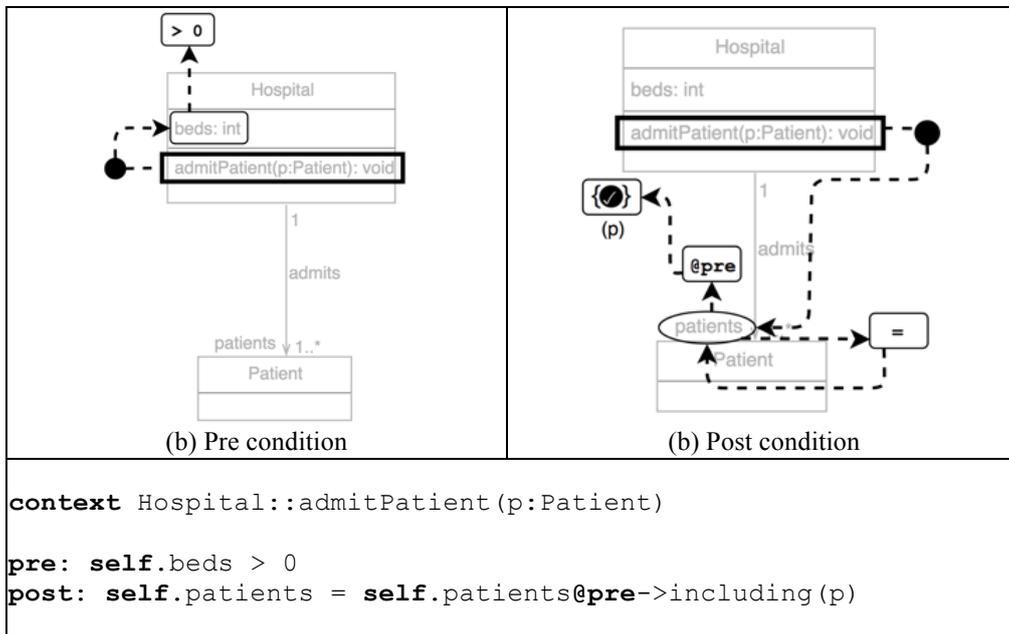


Fig. 3: Pre and post condition example

5 vOCL language Specification

In this section, we introduce the language elements and components using a running example, and provide the language meta-model.

5.1 Running example

Fig. 4 is a model of an *EmergencyRoom* with one Director who supervises a number of nurses of type *ChargeNurse*. These charge nurses supervise a number of nurses of type *Nurse*, each of which can attend to more than one patient. The class nurse has multiple attributes that are inherited by *ChargeNurse*, a subclass of *Nurse*. We use this model to demonstrate vOCL standard operation elements, collection and iteration operations.

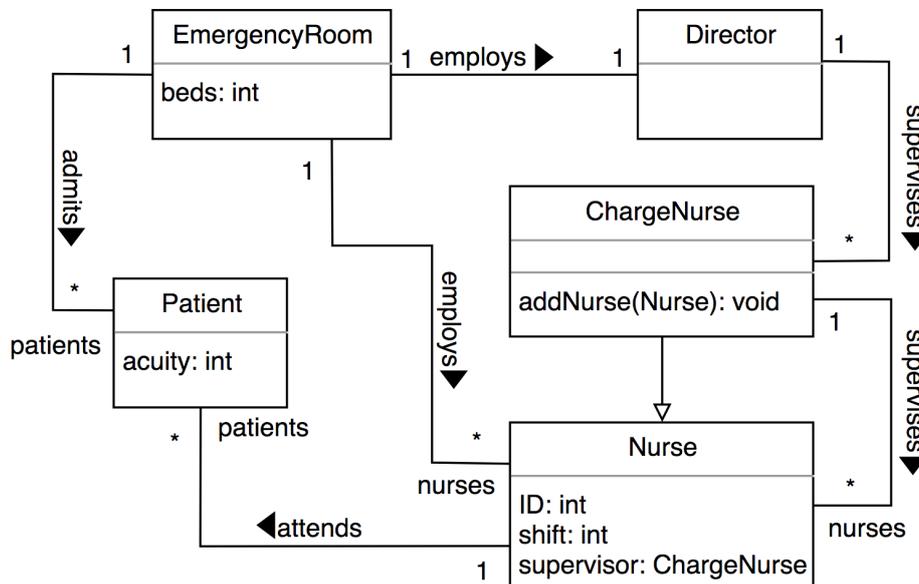


Fig. 4: Emergency Room model

5.2 Standard Operations

Standard operations on collections uses predicate logic to specify invariants. For example, we specify the number of nurses to be greater than the number of patients in the ER. This requires that the size of the set of patients to always be less than the size of the set of nurses.

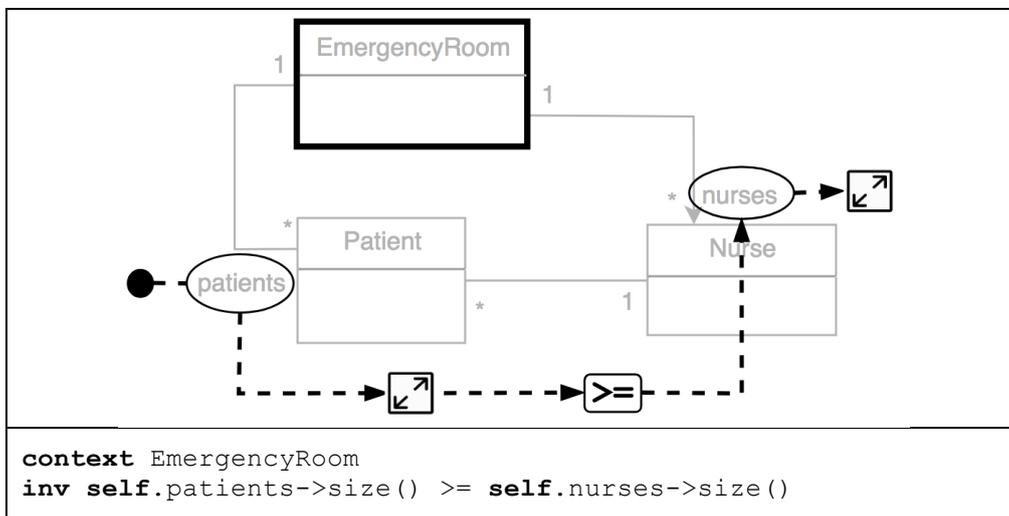


Fig. 5: vOCL constraint demonstrating standard operations

5.3 Collection Operators

Collection Operators uses predicate logic to specify properties of collections. For example, a requirement for the method `addNurse()` may be that at time of completion, the method must have successfully added the new nurse at the end of the “nurses” collection. The OCL constraint

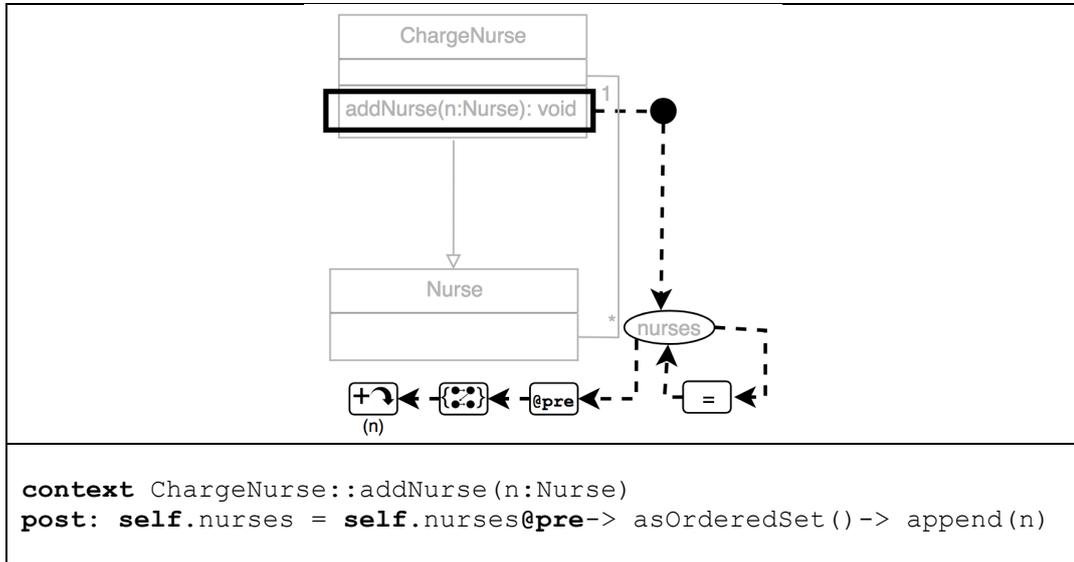


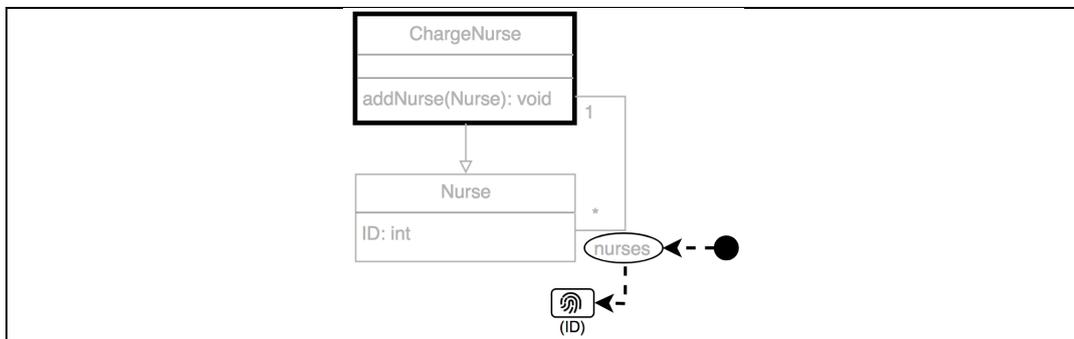
Fig. 6: vOCL constraint demonstrating collection operation

for this requirement is stated in figure 6. It states that a post condition for this method is that the “nurses” collection must be equal to the “nurses” collection before the execution of the method taken as an ordered set.

The append () operation can only be applied to an ordered set or a sequence, which is why that operation is done before appending a new nurse. For the exact guidelines on OCL collection operations, refer to the published OCL specifications [6]

5.4 Iterator Operators

OCL iterator operations on collections are used when we need to iterate over every element one by one for a specific condition. Just as with collection operations, these are predefined iteration operations built on top of the default iterator function. To demonstrate how some of the iterator operators can be used using vOCL consider the following OCL statement that declares that all the nurses under the supervision of a charge nurse must all have unique IDs. The isUnique () operation does all the work of integrating for the user. It iterates over each element one by one to ensure that the attribute given in the operation’s arguments is in fact unique for each member of the collection.



```

context ChargeNurse
inv self.nurses -> isUnique (n:Nurse | n.ID)

```

Fig. 7: vOCL constraint demonstrating isUnique() operation

As a second example, the OCL below uses the forAll() operator to state that for the set of all nurses employed by the ER, all nurses must have the same shift as their supervising charge nurse.

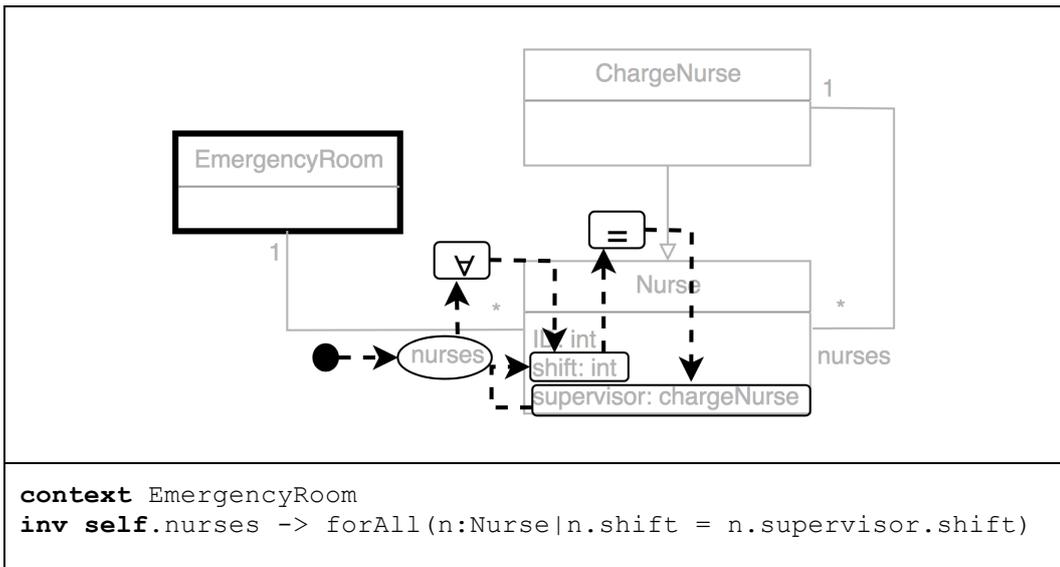


Fig. 8: vOCL constraint demonstrating forall() operation

This constraint is more involved than previous examples. We start by applying the iterator operation forall() to the collection of nurses (the ones employed by the emergency room, as that is the context of our constraint). Next, we see, by following the navigational arrows, that the attribute “shift” must be equal to another integer. So far, we have a constraint that reads: “for all nurses, their shift number must be equal to another integer.” That other integer comes in the right-hand side of the equal operator. Just as OCL uses dot notation to refer to attributes of a class, vOCL allows this same function through the use of directional arrows going from a class object to its attribute. The fact that ChargeNurse is a subclass of Nurse and yet Nurse has a supervisor of type ChargeNurse makes this example relatively complex. Directional arrows coming into and out of the elements should be navigated in a clockwise order.

6 vOCL Meta-Model

vOCL contains various types of visual elements that have a relationship to one another. Aside from that, vOCL is very closely tied to UML and OCL constructs. Below is a meta-model for vOCL that shows the organization of the notation.

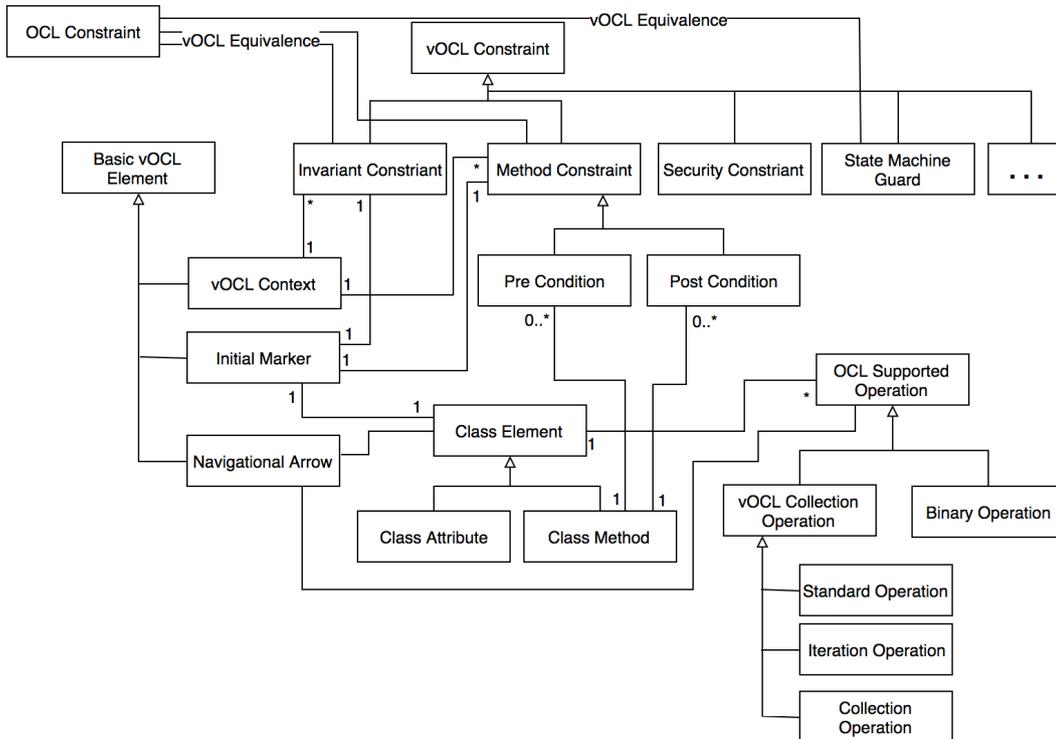


Fig. 9: vOCL Meta-model

The meta-model describes the core constructs of vOCL. As mentioned before, the two main types of constraints used in this paper were invariants, and method pre and post conditions. vOCL constraints can also be extended to create security constraints and state machine guards, but those are outside the scope of this paper and are ultimately potential future work. Current OCL provides support for invariants, method conditions, and state machine guards, making that a direct relationship with the vOCL equivalencies described in this paper. vOCL invariants and method constraints are associated with a vOCL context and an initial marker. Just as in OCL, a single context can support multiple constraints, hence the one to many relationship; however, each individual constraint is associated with one initial marker. The contents of the a vOCL constraint are composed out of class elements and operations on these elements. We take vOCL operations from ones already supported by standard OCL. These operations are applied to class elements through the use of navigational arrows. This model gives an overview of the key constructs of vOCL and their relationship to one another.

7 Limitations

Many visual expression languages have been proposed in the past with almost all of them not gaining traction, arguably with the exception of SDL 22. The proposed visual language suffers from some of important issues that we present in the following.

First, the use of the bolded elements (classes, attributes, etc) to designate a self instance works for a single instance but fails to generalize. Another related limitation is the ability to represent two constraints at the same time on the same UML model. The current proposal assumes only a single constraint. One option to represent multiple constraints is to distinguish them using different colors.

Another important limitation is the ability to realize this proposal in a working modeling tool.

The proposed language extends the UML instance model which could make implementing the proposal in a working tool a challenge. While the proposal is intuitive, it does not consider many facets and needs of implementing the visualization in any modeling tool.

A third limitation is related to the presented meta model. The proposed meta model omits important elements. For example, the Iteration Operation lacks iterators, accumulator, and body. More importantly, and since the proposed vOCL adopts the OCL semantics as-is, a better meta model maybe constructed from the OCL metamodel.

Finally, this proposal uses small and simple OCL statements. It is possible that the visual representation becomes very cumbersome as the context diagram and constraint become larger and more complex.

8 Conclusion and Future Work

One factor limiting OCL broader adoption is in the language textual notation. OCL notation tent to be mathematical and not readily consumable by software engineers. More over, the language design means that its users must constantly shift their focus between the visual UML diagram and the textual OCL notation.

This paper represents work to address this fundamental limitation in the language usability. In this work, we attempt to represent the constraints on top on the UML context diagram. This would minimize or eliminate the need for the users to shift their focus between two different diagrams. vOCL, the proposed notation, guides users through the navigational elements of the language.

There are potentially two concerns with the proposed notation. First, the visual model may become too cumbersome as for more complex constraints. In those situations, the textual notation may in fact be more effective. A second concern relates to the realization of the proposed visual language. vOCL requires that the constraint be modeled on top of the UML context diagram. The implementation of such language will require delicate considerations of the UML model elements and their lay outs.

Our future work includes extending OCL with stochastic and uncertainty elements. There are existing works that extend OCL with probabilities and uncertainties. We plan to build on this work to extend vOCL with those elements.

References

1. Battoni, P, Koch, M, Parisi-Presicce, F, Taentzer, G. Visualization of OCL Using Collaborations. M. Gogolla and C. Kobryb (Eds): UML 2001, LNCS, 2185, pp. 257-271, 2001. Springer-Verlag Berlin Heidelberg 2001.
2. Fish, A, Howse, J, Taentzer, G, Winkleman, J. Two Visualizations of OCL: A Comparison.
3. Cabot, Jordi, and Martin Gogolla. Object constraint language (OCL): a definitive guide. Formal methods for model-driven engineering. Springer Berlin Heidelberg, 2012. 58-90.
4. Warmer, Jos, and Anneke Kleppe. "OCL: The constraint language of the UML." (1999): 10-+.
5. Object Management Group. OMG Unified Modeling Language (UML). Version 2.5. March, 2015.
6. Object Management Group. OML Object Constraint Language (OCL). Version 2.4. February 2014.
7. Pandey, R. K. "Object constraint language (OCL): past, present and future." *ACM SIGSOFT Software Engineering Notes* 36.1 (2011): 1-4.
8. Lodderstedt, Torsten, David Basin, and Jürgen Doser. "SecureUML: A UML-based modeling language for model-driven security." *«UML» 2002—The Unified Modeling Language* (2002): 426-441.
9. Toval, Ambrosio, Víctor Requena, and José Luis Fernández. "Emerging OCL tools." *Software & Systems Modeling* 2.4 (2003): 248-261.

10. Seidewitz, E. (2014, October). UML with meaning: executable modeling in foundational UML and the Alf action language. In *ACM SIGAda Ada Letters*(Vol. 34, No. 3, pp. 61-68). ACM.
11. Papajorgji, P. J., & Pardalos, P. M. (2014). The Object Constraint Language (OCL). In *Software Engineering Techniques Applied to Agricultural Systems* (pp. 121-134). Springer US.
12. Brucker, A. D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., ... & Wolff, B. (2014). Panel discussion: Proposals for improving OCL. In *Proceedings of the MODELS 2014 OCL Workshop (OCL 2014)* (Vol. 1285, pp. 83-99). CEUR-WS. org.
13. Object Management Group. Object Constraint Language™ (OCL™), Version 2.4, Release Date: February 2014. Available: <http://www.omg.org/spec/OCL/2.4/>
14. Brucker, A. D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., ... & Wolff, B. (2014). Panel discussion: Proposals for improving OCL. In *Proceedings of the MODELS 2014 OCL Workshop (OCL 2014)* (Vol. 1285, pp. 83-99). CEUR-WS. org.
15. Bajwa, I. S., Bordbar, B., & Lee, M. G. (2010, October). OCL constraints generation from natural language specification. In *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International* (pp. 204-213). IEEE.
16. Kent, S. (1997, October). Constraint diagrams: visualizing invariants in object-oriented models. In *ACM SIGPLAN Notices* (Vol. 32, No. 10, pp. 327-341). ACM.
17. 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.
18. A. Fish and J. Flower. Investigating reasoning with constraint diagrams. In *Visual Language and Formal Methods, ENTCS*, pages 53–67, Rome, Italy, 2004. Elsevier.
19. Fish, A., & Howse, J. (2004). Towards a default reading for constraint diagrams. *Diagrammatic Representation and Inference*, 1-33.
20. Störrle, H. (2011, September). Expressing model constraints visually with VMQL. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on* (pp. 195-202). IEEE.
21. Fetais, N. (2013). Evaluation of the usability of constraint diagrams as a visual modelling language: theoretical and empirical investigations (Doctoral dissertation, University of Sussex).
22. Ellsberger, Jan, Dieter Hogrefe, and Amardeo Sarma. *SDL: formal object-oriented language for communicating systems*. Prentice Hall, 1997.